Sigurnost i protokoli u razvoju web aplikacija

Baranašić, Mihael

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mihael Baranašić

SIGURNOST I PROTOKOLI U RAZVOJU WEB APLIKACIJA

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mihael Baranašić

JMBAG: 0016152670

Studij: Informacijski i poslovni sustavi

SIGURNOST I PROTOKOLI U RAZVOJU WEB APLIKACIJA

ZAVRŠNI RAD

Mentor:

doc. dr. sc. Matija Novak

Varaždin, rujan 2024

Mihael Baranašić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj završni rad istražuje sigurnost i protokole u razvoju web aplikacija, s posebnim naglaskom na HTTPS. Analizira se povijest razvoja HTTPS-a, vrste napada u nedostatku HTTPS-a te usporedba kriptoalgoritama, uključujući njihove prednosti i slabosti. Rad se bavi usporedbom HTTP-a i HTTPS-a, metodama sprječavanja napada kao što je Man in the Middle, te različitim pristupima implementaciji HTTPS-a u malim i velikim web aplikacijama. Posebna pažnja posvećena je ulozi SSL certifikata u osiguravanju identiteta servera, postupku dobivanja i instalacije certifikata te sigurnosnom sloju HTTP Strict Transport Security (HSTS). Također se razmatraju najbolje prakse za sigurnost TLS konfiguracije te alati i tehnike za testiranje sigurnosti HTTPS-a. Praktični dio uključuje generiranje ključeva i certifikata za Node.js server, pregled modula i alata za implementaciju HTTPS-a te demonstraciju prijelaza Node.js servera s HTTP-a na HTTPS.

Ključne riječi: HTTPS; kriptoalgoritmi; SSL certifikat; TLS; HSTS; web aplikacije; Node.js; Man in the Middle; sigurnost

Sadržaj

1. Uvod 1
2. HTTPS protokol 3
   2.1. Povijest razvoja HTTPS-a 4
      2.1.1. HTTP/0.9 5
      2.1.2. HTTP/1.0 5
      2.1.3. HTTP/1.1 7
      2.1.4. HTTP/2 8
   2.2. Usporedba HTTPS-a i HTTP-a 8
   2.3. HTTP Strict Transport Security 10
3. Kriptoalgoritmi 12
   3.1. Primjena kriptoalgoritama 13
   3.2. Usporedba kriptoalgoritama 13
4. TLS/SSL 15
   4.1. Secure Sockets Layer (SSL) 15
   4.2. Transport Layer Security (TLS) 16
   4.3. Certifikati 18
      4.3.1. Struktura certifikata 19
      4.3.2. Vrste certifikata 19
5. Napadi na web aplikacije 21
   5.1. Vrste napada na web aplikacije 21
   5.2. Detekcija i prevencija napada 22
6. Aplikacija ChatApp 24
   6.1. Funkcionalnosti aplikacije 24
   6.2. ERA model baze podataka 25
   6.3. Pregled korištenih modula 26

6.4. Stranice aplikacije 27
6.4.1. Prijava i registracija 27
6.4.2. Glavna stranica 27
6.4.3. Profil korisnika 28
6.4.4. Glavna stranica – Moderator 28
6.4.5. Dnevnik aktivnosti 29
6.5. Server.js 29
6.5.1. Prikaz i obrada poruka 31
6.5.2. Spremanje i brisanje datoteka 33
6.6. Demonstracija presretanja poruka bez HTTPS-a 36
6.6.1. Hvatanje prometa (tcpdump) 36
6.6.2. Wireshark 37
6.6.3. Pregled prometa 37
6.7. Prijelaz na HTTPS 39
6.7.1. Generiranje ključeva i certifikata 39
6.7.2. Implementacija HTTPS-a na Node.js server 41
6.7.3. Dodatni sigurnosni sloj HSTS 44
6.7.4. Provjera rada HTTPS-a i HSTS-a 45
6.8. Sigurnost podataka sa HTTPS-om 47
7. Zaključak 49
Literatura 50
Popis slika 53
Popis tablica 53
Prilozi 53

1. Uvod

U današnje vrijeme, sigurnost web aplikacija postala je ključni aspekt u razvoju softvera, posebno s obzirom na sve veći broj napada i sigurnosnih prijetnji koje ciljaju osjetljive podatke. Web aplikacije koriste se za razne svrhe, od e-trgovine do financijskih transakcija, i svakodnevno obrađuju velike količine povjerljivih informacija. Stoga je važno osigurati da su te aplikacije zaštićene od neovlaštenog pristupa, manipulacije i presretanja podataka.

Prema Ericu Quintonu [1, str.1], web aplikacija se može definirati kao skup instrukcija koje može interpretirati računalni sustav i koje uključuju podatke, a čije korisničko sučelje radi u pregledniku, dok se logika obrađuje na serveru, tj. udaljenom računalu. Web aplikacije su postale ključne u modernom poslovanju zbog njihove sposobnosti da omogućuju pristup informacijama i uslugama putem interneta [1, str. 1-2]. Ova pristupačnost dolazi s povećanim sigurnosnim rizicima, jer web aplikacije često postaju mete za cyber napade. Povreda sigurnosti može rezultirati gubitkom povjerenja korisnika, financijskim gubicima i štetom za reputaciju organizacije.

Sigurnosni protokoli, kao što je HTTPS, igraju ključnu ulogu u zaštiti podataka i osiguravanju sigurnog okruženja za korisnike. HTTPS ne samo da osigurava šifriranje podataka tijekom prijenosa, već također pruža autentifikaciju servera kroz SSL/TLS certifikate, smanjujući time rizik od phishing napada i drugih sigurnosnih prijetnji. Implementacija HTTPS-a je od suštinskog značaja za svaku web aplikaciju jer šifrira komunikaciju između korisničkog preglednika i servera, čineći podatke nečitljivima potencijalnim napadačima. Ovo je posebno važno za zaštitu osjetljivih informacija, kao što su lozinke, brojevi kreditnih kartica i osobni podaci. Osim šifriranja, HTTPS osigurava da su korisnici povezani s pravim serverom, a ne s lažnim, što je ključno za sprečavanje phishing napada [2].

HTTPS protokol, koji kombinira HTTP s Transport Layer Security (TLS) protokolom, pruža šifriranje podataka između klijenta i servera, osiguravajući povjerljivost i integritet prenesenih informacija. HTTPS je postao standard za web stranice koje zahtijevaju visoku razinu sigurnosti, kao što su bankovne stranice, sustavi za online plaćanje i druge aplikacije koje obrađuju osjetljive podatke.

Važno je pažljivo pohranjivati osjetljive informacije, kao što su prijavni podaci i brojevi kreditnih kartica, te razvijati robusne strategije prijave za ugrađene baze podataka kada je to primjenjivo [1]. U mnogim slučajevima, prva mjera koja se treba poduzeti je šifriranje medija za pohranu. Međutim, osim pohrane, vrlo važna stvar kod sigurnosti web aplikacija je sigurno

prenošenje podataka sa servera do klijenta i obratno. Kod toga nam pomaže HTTPS protokol koji je ključan za prevenciju nekoliko mogućih napada na web aplikaciju i njezine podatke.

Cilj ovog rada je istražiti i analizirati sigurnosne protokole u razvoju web aplikacija, s posebnim naglaskom na HTTPS. Rad će obuhvatiti povijest razvoja HTTPS-a, vrste napada koji se događaju u nedostatku HTTPS-a te usporedbu kriptoalgoritama uključenih u HTTPS. Također će se razmotriti metode sprječavanja napada poput Man in the Middle (MitM) i različiti pristupi implementaciji HTTPS-a u malim i velikim web aplikacijama. Praktični dio rada uključivat će generiranje ključeva i certifikata za Node.js server, pregled modula i alata za implementaciju HTTPS-a te demonstraciju prijelaza s HTTP-a na HTTPS.

Sigurnost web aplikacija je složeno i dinamično područje koje zahtijeva stalno praćenje i prilagođavanje novim prijetnjama. Kroz ovaj rad nastojat će se pružiti dubinsko razumijevanje sigurnosnih protokola te praktične smjernice za implementaciju sigurnih web aplikacija. S obzirom na brzinu razvoja tehnologije i sve sofisticiranije metode napada, ovakva istraživanja su od vitalne važnosti za očuvanje sigurnosti i integriteta web aplikacija u budućnosti.

2. HTTPS protokol

Hypertext Transfer Protocol Secure (HTTPS) je protokol za sigurnu komunikaciju preko računalne mreže, on je glavni protokol korišten na internetu. HTTPS kombinira Hypertext Transfer Protocol (HTTP) s Transport Layer Security (TLS) protokolom kako bi pružio šifriranje i sigurnost podataka između klijenta i servera [3]. Korištenje HTTPS-a je postalo standard za web stranice koje zahtijevaju visoku razinu sigurnosti, poput bankovnih stranica, e-trgovine, i drugih web aplikacija koje obrađuju osjetljive podatke.

HTTPS je sigurnija verzija HTTP-a koja enkriptira poruke u tranzitu koristeći TLS (ranije poznat kao SSL). HTTPS dodaje tri važne komponente HTTP porukama [4]:

- Enkripcija: Poruke ne mogu biti pročitane od strane trećih strana dok su u tranzitu.
- Integritet: Poruka nije izmijenjena u tranzitu jer je cijela enkriptirana poruka digitalno potpisana, a taj potpis je kriptografski verificiran prije dekripcije.
- Autentifikacija: Server je onaj kojem ste namjeravali poslati podatke.

HTTPS koristi enkripciju javnim ključem, omogućujući serverima da pružaju javne ključeve u obliku digitalnih certifikata kada se korisnici prvi put povezuju. Preglednik enkriptira poruke koristeći ovaj javni ključ, koji samo server može dekriptirati jer jedino on posjeduje odgovarajući privatni ključ. Ovaj sustav omogućava sigurnu komunikaciju s web stranicama bez potrebe za unaprijed poznatim zajedničkim tajnim ključem, što je ključno za sustav kao što je internet, gdje se nove web stranice i korisnici pojavljuju i nestaju svake sekunde [4], [5, str. 29].

Digitalne certifikate izdaju i digitalno potpisuju različite certifikacijske agencije (CAs) koje preglednik smatra pouzdanima, što omogućava autentifikaciju da je javni ključ za server s kojim se povezujete. No, veliki problem s HTTPS-om je što on samo potvrđuje da se povezujete s tim serverom, a ne i da je taj server pouzdan. HTTPS stranice obično prikazuju zeleni lokot u web preglednicima, što mnogi korisnici pogrešno tumače kao znak sigurnosti, ali je to zapravo samo sigurna enkripcija [5, str. 29]. Više o ovim certifikatima biti će rečeno u kasnijim poglavljima.

Prema IETF RFC 2818 [4], HTTPS također omogućuje korištenje sigurnih socketa kako bi se osigurala sigurnost i integritet komunikacije. Ovaj standard opisuje kako HTTPS koristi kombinaciju SSL/TLS i HTTP kako bi pružio sigurnu vezu. HTTP poruke unutar HTTPS veze mogu se smatrati sigurnima jer je komunikacijski kanal enkriptiran i verificiran od strane CA.

No, potrebno je paziti na sigurnosne ranjivosti i redovito ažurirati sigurnosne protokole kako bi se osigurala maksimalna zaštita podataka.

Pollard [5, str. 28] govori kako je HTTPS izgrađen oko HTTP-a i gotovo je bez šava za sam HTTP protokol. Hostiran je na različitom portu prema zadanim postavkama (port 443 umjesto 80 za standardni HTTP), i ima drugačiju URL shemu (https:// umjesto http://), ali ne mijenja temeljno način na koji se HTTP koristi u smislu sintakse ili formata poruka, osim za enkripciju i dekripciju. Kada se klijent poveže na HTTPS server, prolazi kroz fazu pregovora (ili TLS handshake) u kojoj server pruža javni ključ, klijent i server dogovaraju metode enkripcije, a zatim pregovaraju zajednički enkripcijski ključ za buduću upotrebu. Nakon uspostavljanja HTTPS sesije, razmjenjuju se standardne HTTP poruke, koje klijent i server enkriptiraju prije slanja i dekriptiraju po primitku.

Kao njegov predak HTTP je vrlo važan protokol, te je važno poznavati sam HTTP protokol, kako radi i kako se razvijao. Prema Pollardu [5, str. 9] Hypertext Transfer Protocol ili HTTP, kao što je navedeno u nazivu, je prvotno bio namijenjen prijenosu hipertekstualnih dokumenata (dokumenata koji sadrže poveznice na druge dokumente), a prva verzija nije podržavala ništa osim ovih dokumenata. Kako se Internet ubrzano razvijao tako su se razvijali i njegovi protokoli. HTTP je ubrzo imao mogućnost prijenosa drugih tipova datoteka kao što su slike, audio i video zapisi.

HTTP je u suštini protokol zahtjev-odgovor. Pollard [5, str. 15-16] opisuje HTTP zahtjev-odgovor tako da web preglednik šalje zahtjev web serveru koristeći HTTP sintaksu, a server odgovara porukom koja sadrži traženi resurs. Ključ uspjeha HTTP-a je njegova jednostavnost. Međutim, ta jednostavnost može predstavljati problem za HTTP/2, koji žrtvuje dio te jednostavnosti radi veće učinkovitosti.

2.1. Povijest razvoja HTTPS-a

Početak HTTP-a krenuo je 1989. godine kada su Tim Berners-Lee i njegov tim u istraživačkoj organizaciji CERN željeli postići povezanu mrežu računala, kako bi se lakše povezali preko interneta, te klikom na poveznicu otvorili poslani dokument [5, str. 4]. Berners je objavio prijedlog za izgradnjom sustava koji će tako povezivati dokumente, te je krenuo sa izgradnjom prvog takvog web poslužitelja (WorldWideWeb) koji se temeljio na HTTP-u.

Prva razvijena verzija i prva verzija koja je imala opisanu specifikaciju je HTTP/0.9. Ovaj protokol stvara konekciju preko Transmission Control Protocol/Internet Protocol servisa ili skraćeno TCP/IP servisa, na server [5, str. 6-7].

2.1.1. HTTP/0.9

Specifikacija ovog protokola je bila vrlo mala (manje od 700 stranica) [5, str. 15-16]. Jedan redak ASCII teksta treba biti poslan, sastavljen od GET zahtjeva, adrese dokumenta (bez razmaka), te povratak na početak reda i novi red (povratak na početak reda može biti opcionalan). Server odgovara porukom u HTML formatu, koju definira kao "tok bajtova ASCII znakova". Pollard [5, str. 16] također navodi da je poruka završena zatvaranjem veze od strane servera, što nam pokazuje jednostavnost HTTP-a, ali i problem nedostatka podrške za praćenje stanja što otežava razvoj složenijih web aplikacija koje zahtijevaju održavanje korisničkih sesija i drugih stanja između zahtjeva.

GET sekcija/stranica.html je jedina moguća naredba u HTTP/0.9, gdje se „sekcija/stranica.html“ može promijeniti, ali je sintaksa fiksna [6]. Nije postojao koncept HTTP zaglavlja niti podrška za druge vrste medija poput slika. Iznenađujuće je kako se iz ovog jednostavnog zahtjev-odgovor protokola, prvotno zamišljenog za lakši pristup informacijama unutar istraživačkog instituta, ubrzo razvila medijski bogata Svjetska mreža koja je danas neizostavan dio svakodnevnog života. Već u početku, Berners-Lee je svoju inovaciju nazvao WorldWideWeb, što pokazuje njegovu viziju o globalnom dosegu projekta i planovima da postane globalni sustav [5, str. 4].

2.1.2. HTTP/1.0

Ova verzija HTTP-a je prva verzija koja je dobila, u početku, neformalnu specifikaciju Request for Comments (RFC) od Radnog tijela za razvoj interneta (IETF, eng. Internet Engineering Task Force. Dokument za ovu verziju protokola je RFC 1945 [7], u njemu je napisano „Ovaj dopis pruža informacije za internetsku zajednicu. Ovaj memorandum ne specificira internetski standard bilo koje vrste“, no svejedno je ovaj dopis dodao neke od ključnih značajki za HTTP/1.0.

RFC 1945 proširuje osnovnu funkcionalnost (GET zahtjev) dodavanjem HTTP zaglavlja koja omogućuju detaljnije komuniciranje između klijenta i servera, te dodavanjem POST zahtjeva koji omogućava slanje podataka na server u tijelu zahtjeva, a ne samo putem usklađenog lokatora sadržaja (URL-a, eng. Uniform Resource Locator) [7]. Osim proširenja funkcionalnosti dodana je i podrška za prijašnju verziju HTTP/0.9, kao i način numeriranja verzija (<velika promjena>.<manja promjena>) i troznamenkasti kod odgovora koji pokazuje je li odgovor bio uspješan (kao što je najpoznatiji 404 – not found) [5], [7].

HTTP/1.0 je zadržao GET metodu sličnu onoj iz HTTP/0.9, ali dodao je mogućnost korištenja zaglavlja kako bi se omogućio uvjetovani GET (zahtjev za resursom samo ako je

promijenjen od zadnjeg dohvaćanja). Također, korisnici su sada mogli dohvaćati više od hipertekstualnih dokumenata, uključujući slike, videozapise i druge medije. Metoda HEAD omogućila je klijentima dobivanje svih meta informacija o resursu bez preuzimanja samog resursa. POST metoda omogućila je klijentima slanje podataka serveru, što je posebno korisno za obrasce na web stranicama [5], [7].

Ono što je bilo prijeko potrebno promijeniti kod HTTP protokola je to da je GET metoda dozvoljavala slanje podataka preko URL-a, što nije prikladno za slanje osjetljivih podataka. Prema Pollardu [5, str. 17] POST metoda je prikladnija za slanje osjetljivih podataka jer nisu vidljivi u URL-u, ali treba biti oprezan kod slanja podataka putem nezaštićenog HTTP-a umjesto HTTPS-a.

Protokol HTTP/1.0 je također omogućio dodavanje prilagođenih zaglavlja u zahtjeve, čime se omogućava proširenje funkcionalnosti protokola bez potrebe za ažuriranjem verzije. Zaglavlja su specificirana s nazivom zaglavlja, dvotočkom i sadržajem zaglavlja, pri čemu je naziv zaglavlja neosjetljiv na velika i mala slova. Višestruka zaglavlja istog tipa mogu se poslati kao zasebne linije ili kao jedan redak s vrijednostima odvojenim zarezima. Protokol je dizajniran da bude proširiv, ali prilagođena zaglavlja možda neće biti prepoznata od strane primatelja i mogu biti ignorirana, dok bi standardna zaglavlja trebala biti obrađena od strane servera koji je u skladu s HTTP/1.0 specifikacijom [5], [7].

Neki od definiranih troznamenkastih kodova u RFC 1945 HTTP/1.0 specifikaciji su [7]:

- 1xx - Informativni kodovi
    - 100 - Continue: Klijent može nastaviti s slanjem zahtjeva. (Ovaj kod nije definiran u HTTP/1.0; uveden je u HTTP/1.1.)
- 2xx - Uspješno obrađeni zahtjevi
    - 200 - OK: Zahtjev je uspješno obrađen i odgovor je poslan.
    - 201 - Created: Zahtjev je uspješno obrađen i novi resurs je kreiran.
    - 202 - Accepted: Zahtjev je prihvaćen za obradu, ali obrada još nije završena.
    - 204 - No Content: Zahtjev je uspješno obrađen, ali nema sadržaja za vraćanje.
- 3xx - Preusmjeravanja
    - 301 - Moved Permanently: Resurs je trajno premješten na novu URL adresu.
    - 302 - Found: Resurs je privremeno premješten; koristi se nova URL adresa za privremeni pristup.
- 4xx - Greške u zahtjevu

    - 400 - Bad Request: Server ne može obraditi zahtjev zbog loše sintakse.
    - 401 - Unauthorized: Zahtjev zahtijeva autentifikaciju korisnika.
    - 403 - Forbidden: Server razumije zahtjev, ali odbija njegovo izvršenje.
    - 404 - Not Found: Resurs nije pronađen na serveru.
- 5xx - Greške na serveru
    - 500 - Internal Server Error: Server je naišao na nepoznatu grešku.
    - 501 - Not Implemented: Server ne podržava funkcionalnost potrebnu za ispunjenje zahtjeva.
    - 502 - Bad Gateway: Server je primio nevažeći odgovor od udaljenog servera.

Iako je značajno proširio mogućnosti protokola, HTTP/1.0 služio više kao privremeni dokument koji je odražavao postojeću praksu bez formalne standardizacije svih novih funkcionalnosti zbog čega je bilo potrebno napraviti novu verziju HTTP/1.1 [5, str. 22].

2.1.3. HTTP/1.1

HTTP/1.1 predstavlja značajno poboljšanje u odnosu na HTTP/1.0, rješavajući mnoge njegove nedostatke i dodajući nove značajke za poboljšanje performansi, pouzdanosti i funkcionalnosti web komunikacije.

Jedno od ključnih poboljšanja u HTTP/1.1 je uvođenje stalnih veza. U HTTP/1.0, nova veza se uspostavlja za svaki par zahtjev/odgovor, što je neefikasno i povećava latenciju. HTTP/1.1 omogućava slanje više zahtjeva i odgovora preko jedne veze, smanjujući potrebu za stalnim uspostavljanjem i zatvaranjem veza. Ovo je posebno korisno za web stranice s više resursa (slike, stilove, skripte itd.), poboljšavajući vrijeme učitavanja stranice i smanjujući opterećenje servera [3], [5].

HTTP/1.1 uvodi kodiranje prijenosa u dijelovima, što omogućava serveru da započne slanje odgovora prije nego što zna ukupnu veličinu odgovora. Ovo je korisno za dinamički generirani sadržaj, omogućujući serveru da održava vezu i započne prijenos podataka čim postanu dostupni. Kodiranje prijenosa u dijelovima razbija odgovor u manje dijelove, svaki s vlastitom veličinom, čineći prijenos podataka efikasnijim [3], [5].

U dokumentu RFC 9112 [3] dodane su nove metode kao i proširen skup statusnih kodova u HTTP verziji HTTP/1.1. Uvodi nove metode kao što su PUT i DELETE, poboljšavajući sposobnost protokola da rukuje širim rasponom operacija osim jednostavnog preuzimanja i slanja podataka. Također, novi statusni kodovi poput 100 Continue poboljšavaju komunikaciju između klijenta i servera tijekom obrade zahtjeva.

HTTP/1.1 zahtijeva Host zaglavlje u svim zahtjevima. Ovo zaglavlje specificira naziv domene servera (za virtualni hosting), omogućujući da se više domena hostira na jednoj IP adresi. Ova promjena rješava problem skalabilnosti u HTTP/1.0, gdje server nije mogao razlikovati zahtjeve prema različitim domenama hostiranim na istoj IP adresi [3].

Osim navedenih unapređenja HTTP/1.1 poboljšava i mehanizme keširanja koji poboljšavaju efikasnost mreže i smanjuju opterećenje servera sprječavanjem nepotrebnih prijenosa podataka, te uvodi zahtjeve za bajtovnim rasponima, omogućujući klijentima da zahtijevaju specifične dijelove datoteke što je posebno korisno za nastavak prekinutih preuzimanja i za aplikacije koje trebaju preuzeti samo dio resursa, kao što je video streaming [3], [5].

2.1.4. HTTP/2

HTTP/2 je najnovija verzija Hypertext Transfer Protocola (HTTP) koja donosi značajna poboljšanja u performansama i efikasnosti u odnosu na prethodne verzije. Razvijen kao odgovor na potrebe za bržim i efikasnijim protokolom, HTTP/2 uvodi nekoliko ključnih značajki [8]:

- Multiplexing: Omogućava slanje više zahtjeva preko jedne TCP veze istovremeno, što smanjuje vrijeme čekanja i poboljšava brzinu učitavanja stranica.
- Komprimiranje zaglavlja: HTTP/2 koristi HPACK algoritam za komprimiranje HTTP zaglavlja, smanjujući količinu podataka koji se moraju poslati.
- Server Push: Server može "gurnuti" resurse klijentu prije nego što ih klijent zatraži, smanjujući latenciju.
- Binary protocol: Za razliku od HTTP/1.1 koji koristi tekstualni format, HTTP/2 koristi binarni format, što je efikasnije i manje sklono greškama u interpretaciji.

Prema Pollardu [5] te prema RFC 9113 [8], HTTP/2 je dizajniran da bude kompatibilan s HTTP/1.1, osiguravajući da novi protokol može raditi s postojećim web infrastrukturom uz minimalne prilagodbe.

2.2. Usporedba HTTPS-a i HTTP-a

HTTP (Hypertext Transfer Protocol) i HTTPS (Hypertext Transfer Protocol Secure) su temeljni protokoli koji omogućuju prijenos podataka na webu. Ključna razlika između ova dva protokola leži u sigurnosti.

Sigurnost [1], [5], [9]:

- HTTP prenosi podatke u čistom tekstu, što ih čini ranjivima na presretanje i manipulaciju, dok HTTPS koristi TLS (Transport Layer Security) za šifriranje podataka u tranzitu. To osigurava da korisnički podaci ne mogu biti presretnuti i pročitani od strane neovlaštenih trećih strana.
- HTTPS također pruža autentifikaciju servera putem SSL/TLS certifikata, što dodatno osigurava da se korisnici povezuju s legitimnim serverima.

Performanse [4], [5], [9]:

- Iako HTTPS zahtijeva dodatno vrijeme za uspostavljanje TLS veze (poznato kao TLS handshake), novije verzije TLS-a i optimizacije u HTTP/2 smanjuju ovaj utjecaj, omogućujući brže učitavanje stranica. Nasuprot tome, HTTP ne zahtijeva dodatne korake za enkripciju, što ga čini nešto bržim u jednostavnim uvjetima, ali mnogo manje sigurnim.
- HTTP/2, kao što je prethodno navedeno, dodatno poboljšava performanse HTTPS-a u usporedbi s HTTP/1.1, unatoč dodatnom trošku šifriranja.
- HTTPS može biti sporiji od HTTP-a zbog dodatnog troška enkripcije, te certifikati potpisani od strane poznatih autoriteta koji pružaju veću sigurnost i višu razinu povjerenja, poput OV i EV, mogu biti skupi.

Povjerenje korisnika:

- Korištenje HTTPS-a postalo je standard za web stranice koje obrađuju osjetljive informacije, kao što su bankovne aplikacije i e-trgovine. Prisutnost HTTPS-a, često vizualizirana ikonom zelenog lokota u pregledniku, povećava povjerenje korisnika u sigurnost web stranice.
- Web stranice koje i dalje koriste HTTP sve se više smatraju nesigurnima, posebno s obzirom na inicijative kao što je Google Chrome koji označava HTTP stranice kao nesigurne.
- HTTPS poboljšava SEO (eng. Search engine optimization) rangiranje web stranice signalizirajući pretraživačima da je stranica sigurna.

Uvođenjem HTTPS-a, web aplikacije su značajno unaprijedile svoju sigurnost. Ipak, važno je naglasiti da samo prisustvo HTTPS-a ne garantira potpunu zaštitu. Potrebno je

dosljedno primjenjivati najbolje prakse u konfiguraciji TLS-a te redovito ažurirati sigurnosne protokole kako bi se osigurala maksimalna zaštita podataka.

Kroz usporedbu HTTP-a i HTTPS-a, postaje jasno da je HTTPS ključan korak naprijed u zaštiti korisničkih podataka na internetu. Unatoč nekim nedostacima, poput mogućih složenijih performansi i dodatnih troškova za certifikate, prednosti HTTPS-a daleko nadmašuju ove izazove. Sigurnost korisnika, povjerenje koje web stranica gradi s posjetiteljima, te poboljšanje SEO rangiranja čine HTTPS neizostavnim standardom za moderne web stranice.

Uvođenje naprednih protokola poput HTTP/2 dodatno smanjuje performansne razlike između HTTP-a i HTTPS-a, osiguravajući brže i sigurnije korisničko iskustvo. Kako se sigurnosne prijetnje na internetu nastavljaju razvijati, sigurnosni protokoli poput HTTPS-a moraju pratiti te promjene. Kontinuirana edukacija o sigurnosnim praksama i redovito ažuriranje sigurnosnih certifikata ključni su za održavanje visokih standarda zaštite podataka u budućnosti.

2.3. HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) je sigurnosna politika definirana u RFC 6797 [10] koja omogućava web stranicama da prisile preglednike na korištenje HTTPS veze, umjesto neosiguranog HTTP-a. Implementacija HSTS-a pomaže u zaštiti od napada poput man-in-the-middle (MitM) napada, posebno onih koji ciljaju na downgrade HTTP veze.

Kada je HSTS omogućen, web server šalje posebnu zaglavlje (HTTP header) Strict-Transport-Security koje pregledniku nalaže da sve buduće veze prema toj web stranici moraju koristiti HTTPS. Osim toga, preglednik je dužan zanemariti sve naredbe korisnika da pređe na HTTP. To značajno smanjuje mogućnost da napadač uspješno presretne ili manipulira komunikacijom koristeći neosigurane HTTP veze [10].

Primjer zaglavlja Strict-Transport-Security [10]:

```
Strict-Transport-Security:max-age=31536000;includeSubDomains;preload
```

- max-age parametar definira vrijeme (u sekundama) koliko preglednik treba poštivati ovu politiku. Vrijednost od 31536000 sekundi odgovara jednoj godini.
- includeSubDomains osigurava da se HSTS politika primjenjuje na sve poddomene glavne domene.
- preload omogućava da se domena uvrsti na popis unaprijed učitanih HSTS domena u preglednicima, čime se dodatno smanjuje mogućnost da korisnici uopće pokušaju koristiti HTTP.

U Node.js ovaj dodatan sloj zaštite se jednostavno implementira uz pomoć Express.js frameworka i helmet modula, koji uključuje srednji sloj (middleware) za konfiguraciju HSTS-a [10].

```
const express = require('express');
const helmet = require('helmet');

const app = express();

// Postavljanje HSTS politike
app.use(helmet.hsts({
  maxAge: 31536000,  // 1 godina u sekundama
  includeSubDomains: true,
  preload: true
}));
// ostatak serverskog koda
```

U ovom primjeru, helmet.hsts() funkcija osigurava da sve dolazne HTTP zahtjeve preusmjerava na HTTPS i primjenjuje HSTS zaglavlja na odgovore servera. Ova implementacija pruža zaštitu korisnicima web aplikacije, čak i ako oni pokušaju koristiti nesigurne HTTP veze.

3. Kriptoalgoritmi

Funkcija kriptografije je transformacija (ili kriptiranje) informacija sa ili bez tajnih ključeva za svrhu tajnosti ili autentičnosti [11]. Kriptografija, prema Odedu Goldreichu [12, str. 1], odnosi se na dizajn, definiranje i izgradnju računalnih sustava koji su otporni na pokušaje zlonamjerne zloupotrebe ili manipulacije.

Tradicionalno je kriptografija bila usmjerena na izradu i analizu šifriranih sustava, no od 1970-ih godina proširila se na područja kao što su digitalni potpisi i protokoli za provjeru autentičnosti. Kriptografski sustavi uključuju različite matematičke alate, poput jednosmjernih funkcija, generatora pseudonasumičnih brojeva i zero-knowledge dokaza, koji su neophodni za osiguranje povjerljivosti, integriteta i autentičnosti podataka [12, str. 1-2].

Na temelju ove definicije, kriptoalgoritmi korišteni u TLS protokolu omogućuju sigurnu komunikaciju putem interneta, šifriranjem podataka između klijenta i servera. Ovi algoritmi osiguravaju da podaci ostanu privatni i nepromijenjeni tijekom prijenosa, pružajući visoku razinu sigurnosti potrebnu za zaštitu osjetljivih informacija. Prema Jordstadu [11] postoje tri glavne kategorije enkripcije: kriptografija sa simetričnim ključem, asimetrična kriptografija i hashiranje.

Kriptografija sa simetričnim ključem – Pošiljatelj i primatelj koriste isti ključ za šifriranje i dešifriranje poruka. Pošiljatelj šifrira poruku koristeći zajednički ključ, šalje je primatelju, a primatelj isti ključ koristi za dešifriranje i povratak na originalni tekst. Jedna od glavnih prednosti simetrične kriptografije je njena brzina i učinkovitost, osobito kada se radi o velikim količinama podataka. Međutim, glavni izazov je sigurna distribucija ključeva između strana [12], [13], [14].

Kriptografija s javnim ključem ili asimetrična kriptografija – Ovdje se koriste dva povezana ključa: javni i privatni ključ. Javni ključ može se slobodno dijeliti i koristi se za šifriranje poruka, dok privatni ključ, koji mora ostati povjerljiv, služi za dešifriranje. Asimetrični algoritmi kao što su RSA i ECC pružaju visok stupanj sigurnosti, a privatni ključ ostaje tajan čak i kada je javni ključ poznat. TLS/SSL certifikati često koriste RSA ključeve, a njegova snaga temelji se na složenosti faktorizacije velikih brojeva [12], [13], [15].

Hash-funkcija – U ovom algoritmu nema ključa; umjesto toga, iz otvorenog teksta se izračunava fiksna duljina vrijednosti, što onemogućava povratak na izvorni tekst. Hash funkcije poput SHA-256 pružaju jednosmjernu funkcionalnost, što znači da je izuzetno teško (praktički nemoguće) obrnuto konstruirati originalni podatak iz hash vrijednosti. Hash funkcije često se koriste za provjeru integriteta podataka i autentifikaciju korisnika [12], [13].

3.1. Primjena kriptoalgoritama

Kriptografski algoritmi imaju ključnu ulogu u osiguravanju podataka i komunikacija u različitim industrijama. U e-trgovini, kriptografski algoritmi poput Naprednog standarda šifriranja (eng. *Advanced Encryption Standard,* kraće AES) i RSA štite povjerljive informacije i transakcije između korisnika i trgovaca. RSA se koristi za sigurno dijeljenje ključeva, dok AES šifrira transakcijske podatke kako bi se spriječio neovlašteni pristup [14], [15].

Bankarski sektor koristi AES za šifriranje osjetljivih podataka poput osobnih identifikacijskih brojeva (PIN-ova) i bankovnih informacija, te hash funkcije kao što je SHA-256 za osiguranje integriteta i autentifikaciju korisnika. Sigurna komunikacija putem interneta koristi kriptografske algoritme za zaštitu poruka, gdje RSA šifrira poruke i omogućuje razmjenu ključeva, a AES pruža brzo i učinkovito šifriranje velikih količina podataka. TLS protokol kombinira asimetrične i simetrične algoritme kako bi osigurao siguran prijenos podataka između preglednika i servera [14].

U zdravstvenom sektoru, kriptografski algoritmi poput AES-a i RSA-a osiguravaju šifriranje i zaštitu elektroničkih zdravstvenih zapisa tijekom pohrane i prijenosa. Hash funkcije osiguravaju autentifikaciju korisnika i integritet podataka. U industrijskim sustavima i IoT uređajima, ECC (Elliptic Curve Cryptography) je popularan zbog svoje efikasnosti i manjih resursnih zahtjeva, osiguravajući komunikaciju i autentičnost uređaja [14], [16].

3.2. Usporedba kriptoalgoritama

Postoji više kripto algoritama, ali u ovom poglavlju ćemo pokriti jedne od najpopularnijih kriptoalgoritama.

AES (Advanced Encryption Standard) - AES je algoritam kriptiranja sa simetričnim ključem koji koristi iterativni pristup gdje podaci prolaze kroz više krugova transformacija, uključujući supstitucije, permutacije i miješanje . Algoritam može koristiti ključeve duljine 128, 192 ili 256 bita [11], [16], [17]. AES je vrlo brz i efikasan, idealan za šifriranje velikih količina podataka, te pruža visoku razinu sigurnosti i otpornost na većinu poznatih napada. Najveći problem kod ovog algoritma je taj da ključ mora biti sigurno distribuiran između pošiljatelja i primatelja, što može biti izazovno.

RSA (Rivest-Shamir-Adleman): RSA je asimetrični kripto algoritam koji koristi matematičke probleme faktorizacije velikih brojeva. Javni ključ se koristi za šifriranje, dok je privatni ključ potreban za dešifriranje [11], [15], [17]. Pruža visoku razinu sigurnosti, posebno s duljim ključevima (npr. 2048 bita), te se javnim ključem može slobodno distribuirati bez

ugrožavanja sigurnosti [11], [15], [17]. Jedina mana ovog algoritma je ta da je sporiji u usporedbi sa simetričnim algoritmima, što ga čini manje pogodnim za šifriranje velikih količina podataka [11], [15].

SHA-256 (Secure Hash Algorithm 256-bit): SHA-256 je hash funkcija koja stvara 256-bitnu hash vrijednost iz ulaznog podatka. Ovaj postupak je jednosmjeran, što znači da nije moguće dobiti originalni podatak iz hash vrijednosti [11], [15], [16], [17]. SHA-256 pruža brzu provjeru integriteta podataka i osigurava da podaci nisu izmijenjeni. Iako je rijetko, moguće je da različiti ulazni podaci proizvedu isti hash, što se naziva kolizija [15], [16], [17].

Osim opisanih kriptoalgoritama postoje još i mnogi drugi algoritmi, koji su prikazani na tablici 1, zajedno sa nekim od njihovih bitnih atributa.

Tablica 1: Kriptoalgoritmi [15]

Shema	**Vrsta algoritma**	**Doprinositelj**	**Duljina ključa**	**Krugovi**	**Veličina bloka**
AES	Simetrični	Rijndael	128,192, 256	10 ili 12 ili 14	128 bit
DES	Simetrični	IBM 75	56-bit	16	64 bit
3DES	Simetrični	IBM 78	168, 112 bits	48	64 bit
BLOWFISH	Simetrični	Bruce Schneier 93	128-448 bits	-	64 bit
RC4	Simetrični	Ronald Rivest 87	40-128 bits	-	-
RSA	Asimetrični	Rivest, Shamir, Adleman 77	1024	1	Minimalno 512 bitova
DSA	Asimetrični	NIST 91	-	-	-
Diffie-Hellman	Asimetrični	Diffie, Hellman 76	-	-	-
El-Gamal	Asimetrični	Elgamal 84	-	-	-
Paillier	Asimetrični	Paillier 99	-	-	-
MD5	Sažimanje/Heshiranje	Rivest 91	128	-	512 bit
MD6	Sažimanje/Heshiranje	Prof. Rivest 08	-	-	-
SHA	Sažimanje/Heshiranje	NIST 95	160	-	-
SHA256	Sažimanje/Heshiranje	-	256	-	32 bit

Tablica 1 prikazuje različite vrste kriptoalgoritama, uključujući njihove karakteristike poput duljine ključa, broja krugova i veličine bloka. Ona također ističe doprinos određenih znanstvenika i institucija u razvoju ovih algoritama. AES i RSA su među najpoznatijim algoritmima s dugom poviješću primjene, dok su algoritmi poput BLOWFISH-a i RC4-a poznati po svojoj brzini i prilagodljivosti. Sažimanje algoritmi, poput SHA-256, ključni su za osiguranje integriteta podataka, dok asimetrični algoritmi poput Diffie-Hellmana omogućuju sigurno dijeljenje ključeva u nesigurnim okruženjima [15].

4. TLS/SSL

TLS (Transport Layer Security) i njegov prethodnik SSL (Secure Sockets Layer) su ključni kriptografski protokoli koji pružaju sigurnu komunikaciju preko računalnih mreža. Ovi protokoli ne samo da omogućuju povjerljivost podataka, već također osiguravaju integritet i autentifikaciju između klijenta i servera. Drugim riječima, TLS/SSL protokoli pružaju zaštitu podataka u tranzitu, što je posebno kritično za osjetljive informacije poput financijskih transakcija, privatnih poruka, te prijenosa podataka putem aplikacija kao što su e-pošta i VoIP. Sigurnost ovih protokola temelji se na sofisticiranim tehnikama šifriranja, koje osiguravaju da podaci ostanu nečitljivi za neovlaštene strane tijekom prijenosa putem nesigurnih mreža poput interneta [18], [19].

Kako Pollard [5] napominje, „Internet je, kao što ime sugerira, mreža računala, a ne sustav od točke do točke“, što znači da podaci, poput onih s kreditnih kartica, često prolaze kroz mnoge posredne točke prije nego stignu na svoje odredište. Thomas [19] dodatno objašnjava da se podaci kreditne kartice prilikom online transakcija šalju kroz mrežu računala, pri čemu bilo koje od tih računala može presresti te informacije. Upravo iz tog razloga SSL i TLS protokoli su razvijeni kako bi osigurali da su takve osjetljive informacije šifrirane, čineći ih neupotrebljivima čak i ako dospiju u krive ruke.

4.1. Secure Sockets Layer (SSL)

SSL je prvotno razvijen od strane tvrtke Netscape sredinom 1990-ih kako bi se osigurao prijenos podataka putem interneta. Protokol šifrira podatke na transportnom sloju, osiguravajući da informacije poput brojeva kreditnih kartica, vjerodajnica za prijavu i drugih osjetljivih podataka ostanu povjerljive tijekom prijenosa. SSL koristi kombinaciju simetrične i asimetrične kriptografije kako bi osigurao komunikaciju između dviju krajnjih točaka [2], [20].

SSL je strukturiran u dva glavna sloja, pri čemu svaki sloj koristi usluge nižeg sloja i pruža funkcionalnost višim slojevima. Prvi je SSL sloj zapisa, koji osigurava povjerljivost, autentičnost i zaštitu od ponavljanja podataka preko pouzdanog transportnog protokola kao što je TCP. Iznad sloja zapisa nalazi se SSL protokol rukovanja, koji je zadužen za razmjenu kriptografskih ključeva i sinkronizaciju kriptografskog stanja između dviju krajnjih točaka. Nakon uspješnog rukovanja i uspostavljanja sigurnosnih parametara, osjetljivi podaci aplikacije mogu se sigurno slati preko SSL sloja zapisa [20].

Razvoj SSL-a prošao je kroz nekoliko verzija, od kojih su najznačajniji SSL 2.0 i SSL 3.0. SSL 1.0, prva verzija SSL-a, nikada nije bila javno objavljena zbog brojnih sigurnosnih

nedostataka i mana [21]. SSL 2.0, prva javno dostupna verzija, imala je nekoliko značajnih sigurnosnih propusta koji su brzo doveli do razvoja SSL 3.0 verzije. SSL 3.0 je riješio mnoge probleme prethodne verzije i dizajniran je za pružanje povjerljivosti, autentičnosti i zaštite od ponovnog korištenja podataka putem pouzdanog transportnog protokola kao što je TCP. Sloj zapisa u SSL 3.0 pruža ove sigurnosne usluge, dok SSL protokol rukovanja, smješten iznad sloja zapisa, inicijalizira i sinkronizira kriptografsko stanje između dviju krajnjih točaka[2], [20].

Prema RFC 6101 [22], SSL protokol rukovanja koristi nekoliko ključnih koraka za osiguranje komunikacije. Prvo, klijent šalje popis podržanih algoritama šifriranja serveru. Server potom odabire algoritam i šalje svoj certifikat klijentu. Nakon toga slijedi razmjena ključeva i generiranje sesijskog ključa, koji će se koristiti za šifriranje podataka u daljnjoj komunikaciji. Ovaj postupak rukovanja osigurava da su podaci šifrirani i zaštićeni od presretanja tijekom cijele sesije. SSL protokol rukovanja također koristi digitalne potpise kako bi osigurao autentičnost i integritet razmijenjenih ključeva, smanjujući rizik od napada Man-in-the-Middle (MITM) [2].

SSL je postavio temelje za kasnije verzije sigurnosnih protokola, uključujući TLS, koji je naslijedio SSL kao sigurnosni standard za internetske komunikacije. Iako su novije verzije TLS-a uvelike nadmašile SSL u pogledu sigurnosti i performansi, SSL ostaje važan dio povijesti internetske sigurnosti i temeljni koncept na kojem su izgrađeni moderni sigurnosni protokoli.

4.2. Transport Layer Security (TLS)

Oppliger [18] opisuje TLS (Transport Layer Security) kao „Klijent/server protokol koji je smješten iznad pouzdanog protokola transportnog sloja, poput TCP-a u slučaju TCP/IP protokolskog skupa, i koji se sastoji od istih dvaju slojeva i protokola kao i SSL protokol“. TLS je nasljednik SSL-a, razvijen od strane IETF-a (Internet Engineering Task Force) kako bi pružio poboljšanu sigurnost i interoperabilnost u internetskim komunikacijama. Prva verzija TLS-a, TLS 1.0, objavljena je 1999. godine, temeljila se na SSL 3.0, ali je uključivala niz sigurnosnih poboljšanja i podršku za novije kriptografske algoritme. Kroz godine, TLS je kontinuirano evoluirao, a svaka nova verzija donijela je robusnije sigurnosne značajke i optimizacije performansi, što ga je učinilo standardom za sigurne komunikacije na internetu [2], [23].

TLS protokol je strukturalno identičan SSL protokolu, s razlikom što prefiks „SSL“ u nazivima protokola zamijenjen je s „TLS“ [21]. U RFC 8446 [24] se opisuje da kao i SSL, TLS se sastoji od dva sloja: nižeg sloja TLS sloja zapisa (record protocol) i višeg sloja koji uključuje četiri podprotokola: TLS protokol za promjenu šifri (change cipher spec protocol), TLS protokol

upozorenja (alert protocol), TLS protokol rukovanja (handshake protocol) i TLS protokol prijenosa podataka aplikacije (application data protocol). Ovi podprotokoli omogućuju siguran prijenos podataka, pregovaranje sigurnosnih parametara i osiguranje integriteta komunikacije. Na primjer, TLS sloj zapisa fragmentira, po potrebi komprimira i kriptografski štiti podatke više razine protokola, koristeći strukture podataka TLSPlaintext, TLSCompressed i TLSCiphertext.

TLS protokol rukovanja jedan je od ključnih elemenata TLS-a, odgovoran za uspostavu sigurne sesije između klijenta i servera. Tijekom ovog procesa, server se autentificira prema klijentu (s opcionalnom autentifikacijom klijenta), dogovaraju se metode šifriranja (cipher suites) te se razmjenjuju kriptografski ključevi. U kasnijim fazama, TLS sloj zapisa osigurava da svi podaci preneseni između klijenta i servera ostanu šifrirani i zaštićeni od manipulacije. Također, TLS protokol koristi mehanizam sesija i konekcija, gdje više konekcija može biti povezano s jednom sesijom. Ove konekcije koriste četiri stanja: trenutno stanje za čitanje i pisanje te očekivano stanje za čitanje i pisanje, slično kao i u SSL protokolu. Sigurnosni parametri definirani su na razini konekcije i uključuju, između ostalog, algoritam PRF (Pseudo-Random Function), uveden u TLS 1.2 [18], [24].

RFC 8446 [24] opisuje kako TLS protokol pruža tri ključne usluge za sve aplikacije koje se izvode iznad njega: šifriranje, autentifikaciju i integritet podataka. Šifriranje omogućuje maskiranje podataka poslanih između dvaju hostova, autentifikacija verificira valjanost priloženih identifikacijskih materijala, a integritet osigurava otkrivanje manipulacija i krivotvorenja poruka. Kako bi uspostavili kriptografski sigurnu komunikaciju, sugovornici se moraju složiti oko korištenih šifri i ključeva za šifriranje podataka. TLS protokol definira niz rukovanja za ovu razmjenu, koristeći asimetričnu kriptografiju koja omogućuje sugovornicima da dogovore zajednički tajni ključ bez prethodnog poznavanja, i to preko nešifriranog kanala [19].

4.2.1. Najbolje prakse TLS konfiguracije

Za sigurnu TLS konfiguraciju, RFC 8446 navodi ove najbolje prakse [24]:

- Onemogućite zastarjele verzije: Podržavajte samo TLS 1.2 i 1.3; onemogućite SSL i starije verzije TLS-a.
- Koristite sigurne algoritme: Odaberite moderne i jake algoritme poput AES-256 i SHA-256; izbjegavajte RC4, DES i slabije enkripcije.
- Implementirajte Perfect Forward Secrecy (PFS): Koristite DHE ili ECDHE za razmjenu ključeva kako biste osigurali sesije.
- Koristite snažne certifikate: Koristite certifikate s ključevima od najmanje 2048 bita, izdane od pouzdanog CA-a, i redovito ih obnavljajte.

- Implementirajte HSTS: Osigurajte da web stranice koriste samo HTTPS veze kako bi se spriječili downgrade napadi.
- Redovito ažurirajte softver: Održavajte softver ažuriranim kako biste otklonili poznate ranjivosti.
- Konfigurirajte servere za sigurnost: Podržavajte samo sigurne protokole i algoritme.
- Koristite IDS/IPS: Implementirajte sigurnosne alate za otkrivanje i sprječavanje napada, uključujući "man in the middle" napade.

4.3. Certifikati

Prema Stephen Thomasu u *SSL & TLS Essentials: Securing the Web* [19] „certifikati s javnim ključem su digitalni ekvivalent vozačke dozvole. Iako certifikati mogu pripadati računalnim sustavima umjesto pojedincima, dijele tri važne karakteristike s vozačkim dozvolama“. Certifikati sadrže ključne informacije o subjektu, kao što su ime subjekta i javni ključ, te su izdani od strane pouzdanih organizacija poznatih kao Certifikacijski Autoriteti (CA).

Prema RFC 9608 [25], certifikati također uključuju specifična polja koja omogućuju verifikaciju autentičnosti, kao što su izdavač (issuer), predmet (subject) i razdoblje valjanosti (validity period) [24]. Ovi elementi omogućuju pouzdanu identifikaciju subjekta i osiguravaju da certifikat ima važeći i povjerljiv status.

Prema izvoru Networking 101 [2] kada klijent započne komunikaciju s serverom putem HTTPS-a, server šalje svoj certifikat klijentu kao dio procesa uspostave sigurne veze. Ovaj certifikat sadrži javni ključ servera, informacije o izdavaču certifikata (certifikacijski autoritet, CA), te informacije o vlasniku certifikata (serveru). Klijent koristi javni ključ iz certifikata kako bi sigurno šifrirao podatke koji se šalju serveru, osiguravajući da ih samo server može dešifrirati koristeći svoj privatni ključ.

Certifikati se koriste za uspostavu sigurne komunikacije putem SSL/TLS protokola, pružajući sigurnost podataka u tranzitu između klijenta i servera. Oni su temelj sigurnosne infrastrukture na internetu, omogućujući šifriranje podataka, autentifikaciju sudionika u komunikaciji i zaštitu integriteta podataka [19].

4.3.1. Struktura certifikata

Certifikati prema RFC 9608 [25] i Thomasu [19] sadrže nekoliko ključnih elemenata koji omogućuju njihovu funkcionalnost:

- Javni ključ (Public Key): Ovo je kriptografski ključ koji pripada subjektu certifikata i koristi se za enkripciju podataka ili provjeru digitalnog potpisa.
- Podaci o subjektu (Subject Information): Ovi podaci uključuju ime i druge identifikacijske informacije o vlasniku certifikata, kao što su organizacija, lokacija i e-mail adresa.
- Podaci o izdavaču (Issuer Information): Ovdje su navedene informacije o certifikacijskom autoritetu (CA) koji je izdao certifikat.
- Razdoblje valjanosti (Validity Period): Certifikati imaju određeni rok trajanja, unutar kojeg su važeći. Nakon isteka, certifikat više ne može biti korišten za provjeru identiteta.
- Digitalni potpis (Digital Signature): Certifikat sadrži digitalni potpis CA-a, koji potvrđuje autentičnost i integritet certifikata.
- Serijski broj (Serial Number): Jedinstveni broj koji identificira certifikat unutar domene izdavača.

Ova struktura omogućuje da se certifikat koristi za provjeru autentičnosti, enkripciju komunikacije i zaštitu integriteta podataka.

4.3.2. Vrste certifikata

Certifikati se mogu klasificirati prema više kriterija, uključujući namjenu, nivo provjere identiteta, i metodu izdavanja. Opplinger opisuje nekoliko vrsta certifikata [18]:

- Certifikati s domenskom validacijom (Domain Validation - DV): Ovi certifikati potvrđuju vlasništvo nad domenom. Izdaju se nakon što vlasnik domene dokaže kontrolu nad domenom, obično putem e-maila ili DNS zapisa. DV certifikati pružaju osnovnu razinu sigurnosti.
- Certifikati s validacijom organizacije (Organization Validation - OV): OV certifikati ne samo da potvrđuju vlasništvo nad domenom, već i provjeravaju identitet organizacije koja je vlasnik domene. To uključuje provjeru pravnog statusa i fizičke lokacije organizacije.

- Certifikati s proširenom validacijom (Extended Validation - EV): EV certifikati pružaju najvišu razinu provjere identiteta. Izdaju se nakon rigoroznog procesa validacije, koji uključuje temeljitu provjeru pravnog statusa, identiteta i fizičke adrese organizacije. Preglednici često označavaju web stranice s EV certifikatima u adresnoj traci posebnim oznakama, kao što je zelena boja.
- Wildcard Certifikati: Ovi certifikati omogućuju zaštitu više poddomena jedne domene s jednim certifikatom. Na primjer, wildcard certifikat za *.example.com može zaštititi www.example.com, mail.example.com, i druge poddomene.
- Certifikati za više domena (Multi-domain SSL Certificates): Ovi certifikati omogućuju zaštitu više potpuno različitih domena unutar jedne SSL sesije. Na primjer, jedan certifikat može pokrivati example.com, example.net, i example.org.

Osim ovih vrsta certifikata postoje i certifikati potpisani vlastitim potpisom (eng. Self-signed certificates). Njih izdaje sam korisnik ili organizacija, bez intervencije vanjskog certifikacijskog autoriteta (CA). Ovi certifikati su često korišteni u testnim okruženjima, razvoju softvera, ili za interne svrhe unutar organizacija. RFC 8446 [24] za ove certifikate napominje da zbog toga što ne pružaju vanjsku provjeru identiteta vlasnika certifikata, korisnici moraju sami odlučiti vjeruju li certifikatu, a većina web preglednika i aplikacija ne prepoznaje self-signed certifikate kao pouzdane, osim ako korisnik ručno ne doda certifikat u sustav povjerenja. Iako nemaju vanjsku validaciju, self-signed certifikati nude istu osnovnu funkcionalnost kao i standardni certifikati, uključujući šifriranje podataka i zaštitu integriteta, te se koriste kada su sigurnost i povjerenje u certifikat kontrolirani unutar zatvorenog sustava ili mreže.

5. Napadi na web aplikacije

Web aplikacije koje koriste samo HTTP umjesto HTTPS-a posebno su ranjive na brojne napade zbog nedostatka enkripcije u komunikaciji. Kada aplikacija komunicira preko HTTP-a, podaci se šalju u otvorenom tekstu, što omogućava napadačima da lako presretnu i izmijene te informacije. Kako navodi Michal Zalewski [26], "korištenje nezaštićene veze kao što je HTTP ostavlja vrata širom otvorena napadačima koji mogu lako manipulirati podacima tijekom prijenosa".

Bruce Potter i Bob Fleck [27] također ističu da je komunikacija preko nezaštićenih kanala, poput HTTP-a, izrazito osjetljiva na različite oblike napada, uključujući presretanje i manipulaciju podacima.

5.1. Vrste napada na web aplikacije

Presretanje podataka (Eavesdropping):

Kada aplikacija koristi HTTP, svaki podatak koji se šalje može biti presretnut od strane napadača koji ima pristup mreži. To uključuje osjetljive informacije poput lozinki, osobnih podataka i drugih povjerljivih informacija. Prema Zalewskom [26], napadi presretanja podataka su posebno opasni u bežičnim mrežama jer napadač može relativno lako dobiti pristup mrežnom prometu. Potter i Fleck [27] dodaju da "nedostatak enkripcije omogućava napadačima da snime i analiziraju cijeli mrežni promet, što dovodi do potencijalnog curenja osjetljivih informacija".

Man-in-the-Middle (MITM) napadi:

Ovi napadi omogućuju napadaču da se postavi između korisnika i web aplikacije, presrećući i mijenjajući komunikaciju. Napadač može preusmjeriti korisnika na lažnu web stranicu ili izmijeniti podatke bez znanja korisnika. MITM napadi su posebno opasni u HTTP okruženjima jer nema enkripcije koja bi spriječila neovlaštenu manipulaciju podacima. Zalewski [26] napominje da su MITM napadi na nezaštićene veze "najjednostavniji i najefikasniji način za kompromitiranje sigurnosti web aplikacija" . S druge strane, Potter i Fleck [27] opisuju kako napadači koriste ove napade kako bi prikupili informacije ili preuzeli kontrolu nad sesijom korisnika, što je posebno štetno u osjetljivim mrežnim okruženjima.

MITM napadi su obično skriveni unutar mrežnih slojeva i često iskorištavaju slabosti u SSL/TLS protokolima. Kassaras [28] napominje kako napadači mogu koristiti složene kombinacije ranjivosti, poput XXE injekcija u web aplikacijama, da bi unaprijedili svoje MITM

napade. Kombiniranjem različitih tehnika, napadači mogu doći do osjetljivih informacija ili čak manipulirati podacima u realnom vremenu. U knjizi se također ističe važnost korištenja vlastitih prilagođenih alata za penetracijsko testiranje, umjesto oslanjanja na standardne komercijalne skenere koji često ne otkrivaju kompleksne prijetnje.

Napadi presretanja sesije (Session Hijacking):

HTTP sesije često nisu dovoljno zaštićene, što omogućava napadačima da preuzmu korisničku sesiju i pristupe podacima ili funkcionalnostima kao da su legitimni korisnik. To je često omogućeno pomoću presretanja kolačića ili drugih podataka sesije. Zalewski [26] objašnjava da "napadi na sesije često prolaze nezapaženo, sve dok napadač ne iskoristi pristup za zlonamjerne aktivnosti" . S druge strane, Potter i Fleck [27] u svojoj knjizi ističu važnost zaštite sesija putem dodatnih sigurnosnih mjera, kao što su sigurni kolačići i tokeni sesije, kako bi se smanjio rizik od ovakvih napada.

Napadi presretanja odgovora (Response Splitting):

Ova vrsta napada koristi ranjivosti u načinu na koji web aplikacije obrađuju HTTP zaglavlja. Napadač može ubrizgati zlonamjerne zaglavlja koja uzrokuju nepredvidivo ponašanje aplikacije, kao što je preusmjeravanje korisnika na zlonamjerne stranice. Zalewski [26] u svojoj knjizi opisuje kako napadači koriste ove ranjivosti za stvaranje uvjeta u kojima korisnici ne mogu razlikovati legitimne stranice od zlonamjernih, što može dovesti do ozbiljnih sigurnosnih problema.

Osim napada vezanih za HTTP, web aplikacije mogu biti izložene i drugim vrstama napada, kao što su SQL injekcija, Cross-Site Scripting (XSS), i Distributed Denial of Service (DDoS) napadi. Ovi napadi ciljaju na ranjivosti u kodu aplikacije ili infrastrukturi te mogu rezultirati krađom podataka, oštećenjem sustava ili prekidom usluga. Prema Zalewskom [26], "SQL injekcije ostaju među najčešćim prijetnjama za web aplikacije, osobito one koje ne provode stroge provjere unosa korisnika". Potter i Fleck [27] također ističu važnost sigurnosnih mjera koje sprečavaju ove napade, uključujući pravilno filtriranje unosa i redovito ažuriranje softvera.

5.2. Detekciija i prevencija napada

Kako bi se smanjio rizik od ovih napada, važno je implementirati sljedeće mjere:

Korištenje HTTPS-a: Implementacija HTTPS-a osigurava da su svi podaci između korisnika i web aplikacije enkriptirani, što značajno smanjuje rizik od presretanja i MITM napada. "Korištenje HTTPS-a je prvi i najvažniji korak u zaštiti web aplikacija," naglašava

Zalewski [26], dodajući da enkripcija pruža osnovnu razinu zaštite koja onemogućuje jednostavne napade presretanja [27].

Praćenje i detekcija anomalija: Sustavi za detekciju upada (IDS) i praćenje anomalija mogu pomoći u otkrivanju neuobičajenih aktivnosti koje mogu ukazivati na napade. Zalewski [26] napominje da "automatizirani alati za detekciju napada mogu biti ključni u sprječavanju velikih incidenata" , dok Potter i Fleck [27] preporučuju redovito praćenje mrežnih aktivnosti kao dio općeg sigurnosnog sustava.

Redovito ažuriranje i zakrpe: Osiguranje da su svi sustavi i aplikacije ažurirani s najnovijim sigurnosnim zakrpama je ključna mjera prevencije protiv poznatih ranjivosti. Zalewski [26] naglašava važnost redovitog održavanja sustava tako što "neodgovarajuća ažuriranja softvera ostavljaju aplikacije izloženima poznatim ranjivostima".

Penetracijsko testiranje: Redovito provođenje penetracijskih testova može identificirati ranjivosti prije nego što ih napadači iskoriste. Kassaras [28] preporučuje penetracijske testove kao dio sveobuhvatne sigurnosne strategije, koja pomaže identificirati i popraviti potencijalne sigurnosne rupe prije nego što postanu stvarne prijetnje .

Kroz ove mjere, web aplikacije mogu značajno povećati svoju otpornost na napade, osiguravajući bolju zaštitu podataka i kontinuiranu dostupnost usluga.

6. Aplikacija ChatApp

ChatApp je web aplikacija koja omogućuje korisnicima razmjenu tekstualnih poruka u stvarnom vremenu. Cilj ove aplikacije je demonstracija važnosti korištenja HTTPS protokola u osiguravanju privatnosti i sigurnosti komunikacije na webu, kao i ilustracija jednostavnosti kojom se podaci mogu presresti kada se koriste nesigurni HTTP protokoli. Aplikacija je razvijena kao dio praktičnog dijela završnog rada, gdje se istražuju prednosti i sigurnosne prednosti prelaska s HTTP-a na HTTPS.

Aplikacija je također razvijena s namjerom da se jasno pokaže kako nešifrirana komunikacija može biti presretnuta od strane zlonamjernih napadača (tzv. Man in the Middle napadi) te kako implementacija HTTPS-a može zaštititi podatke korisnika. U kontekstu završnog rada, ChatApp služi kao konkretan primjer koji će pokazati postupak prelaska s HTTP-a na HTTPS, uključujući generiranje ključeva i certifikata, kao i konfiguraciju Node.js servera za sigurnu komunikaciju.

Kroz ovu aplikaciju će se demonstrirati i kako jednostavno dolazi do sigurnosnih propusta ukoliko se ne koristi HTTPS protokol. Ovaj praktični primjer pomoći će u razumijevanju teorijskih dijelova rada koji se bave sigurnosnim rizicima nesigurnih mreža i važnosti enkripcije u zaštiti osjetljivih podataka. Na taj način, ChatApp ne samo da služi kao alat za komunikaciju, već i kao edukativni primjer koji podržava teoretske aspekte ovog rada.

Sljedeća podpoglavlja će detaljno opisati funkcionalnosti aplikacije, ERA model baze podataka, korištene module, te će biti prikazane skice zaslona kako bi se pružio cjelovit pregled dizajna i logike iza razvoja ove aplikacije.

6.1. Funkcionalnosti aplikacije

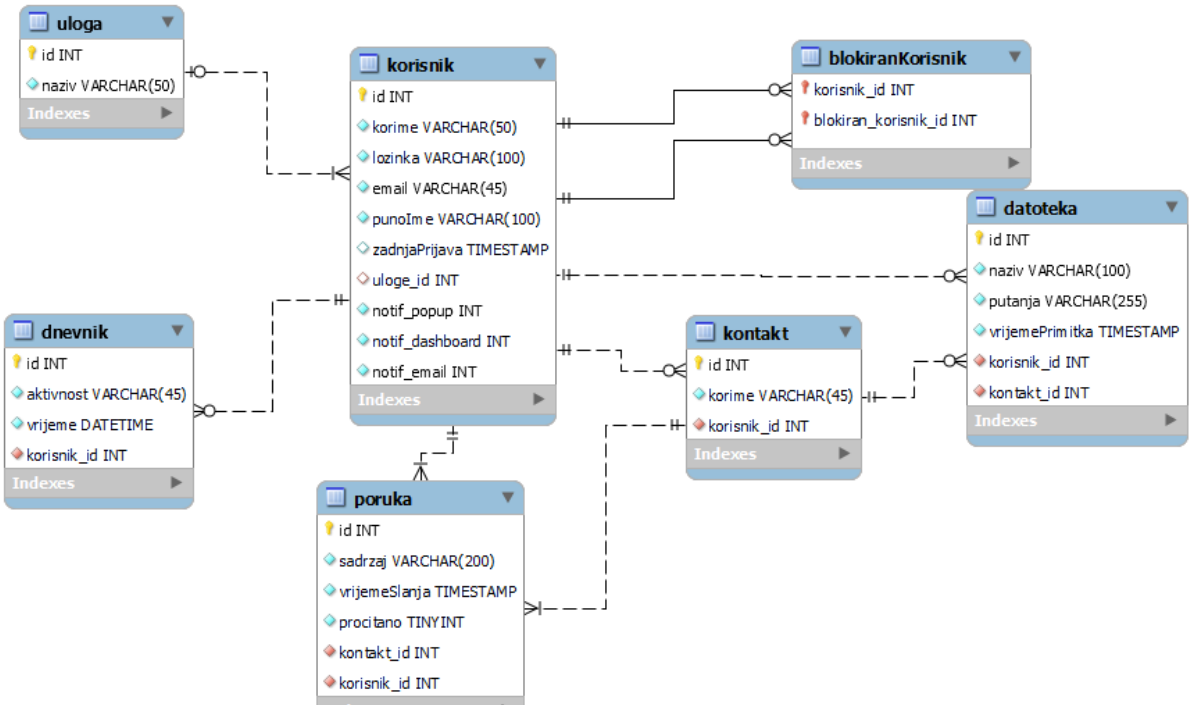
Funkcionalnosti aplikacije ChatApp su podijeljene prema ulogama korisnika:

- Neregistrirani korisnik: Pregled stranice za prijavu i registraciju, prijelaz na stranicu za registraciju, prijava u aplikaciju.
- Registrirani korisnik: Prijavljivanje na svoj korisnički račun, pregled vlastitog profila, pregled liste kontakata, pokretanje novog individualnog razgovora, pregled povijesti poruka, slanje i primanje tekstualnih poruka i datoteka, primanje obavijesti o novim porukama, ažuriranje osobnih podataka, traženje korisnika prema imenu, brisanje vlastitog korisničkog računa, pregled statistika razgovora, prijava neželjenih poruka.

- Administrator: Pregled statistika o korištenju aplikacije, pregled i filtriranje dnevnika aktivnosti, pregled i upravljanje korisničkim računima, primanje upita od moderatora.
- Moderator: Brisanje poruka ili chatova, primanje i odgovaranje na upite korisnika, slanje upita administratoru.

6.2. ERA model baze podataka

Kako bismo mogli spremati podatke o korisnicima, poruke, datoteke i ostale podatke potrebno je napraviti bazu podataka, a kao reprezentacija baze podataka nam može služiti ovaj ERA (eng. Entity Relation Attribute) model napravljen u bazi SQLite pomoću programa MySQL Workbench.



Slika 1: ERA model baze podataka

Kao što je prikazano na Slici 1, veze između tablica omogućavaju efikasno upravljanje ključnim aspektima aplikacije. Na primjer, relacija između tablica Korisnik i Poruka omogućava pohranu svih poruka koje korisnici razmjenjuju. Svaka poruka povezana je s korisnikom putem stranog ključa, čime se osigurava da se sve poruke mogu jednostavno pratiti i prikazati unutar aplikacije, što je ključno za pregled povijesti razgovora i slanje novih poruka.

Relacija između tablica Korisnik i Kontakt omogućava korisnicima da održavaju listu kontakata. Ovo je posebno važno jer aplikacija omogućava započinjanje novih razgovora i upravljanje kontaktima. Korištenje zasebne tablice za kontakte osigurava fleksibilnost u slučaju da jedan korisnik ima više kontakata.

Posebno je zanimljiva relacija između tablica Korisnik i BlokiranKorisnik, koja omogućava implementaciju funkcionalnosti blokiranja korisnika. Ova relacija omogućava više-prema-više povezanost, što znači da svaki korisnik može blokirati više drugih korisnika i biti blokiran od strane više korisnika. Ova struktura je ključna za moderiranje i održavanje sigurnosti unutar aplikacije, jer omogućava korisnicima da kontroliraju s kime žele komunicirati.

Tablice Datoteka i Dnevnik omogućavaju dodatne funkcionalnosti kao što su slanje datoteka unutar chata i praćenje aktivnosti korisnika. Relacije između korisnika i ovih tablica omogućavaju aplikaciji da prati tko je poslao koju datoteku i kada su korisnici obavljali određene radnje, što je bitno za administrativne i sigurnosne funkcionalnosti aplikacije.

Sve u svemu, ovaj model baze podataka omogućava efikasno rukovanje svim bitnim podacima unutar aplikacije ChatApp, pružajući podršku za različite korisničke uloge i osiguravajući da aplikacija može ispravno funkcionirati kako bi zadovoljila potrebe svih korisnika, od običnih korisnika do administratora i moderatora.

6.3. Pregled korištenih modula

Za izradu aplikacije ChatApp korišteno je nekoliko modula dostupnih putem Node Package Managera (npm). Ovi moduli omogućuju razne funkcionalnosti unutar aplikacijeU nastavku su njihovi opisi [29]:

- Express: Izradu i upravljanje HTTP serverom, definiranje ruta te rukovanje HTTP zahtjevima i odgovorima.
- SQLite3: Interakciju s SQLite bazom podataka unutar Node.js aplikacije.
- ws: Implementaciju WebSocket komunikacije koja omogućuje real-time razmjenu poruka.
- express-session: Upravljanje korisničkim sesijama u Express aplikaciji.
- cookie-parser: Parsiranje i upravljanje HTTP kolačićima unutar Express aplikacije.
- multer: Obradu podataka iz obrazaca koji sadrže prijenos datoteka (multipart/form-data).

- dotenv: Upravljanje okolišnim varijablama iz .env datoteke.
- nodemailer: Slanje e-pošte iz Node.js aplikacija koristeći različite SMTP poslužitelje.
- crypto: Kriptografske operacije poput hashiranja, enkripcije i dekripcije podataka.

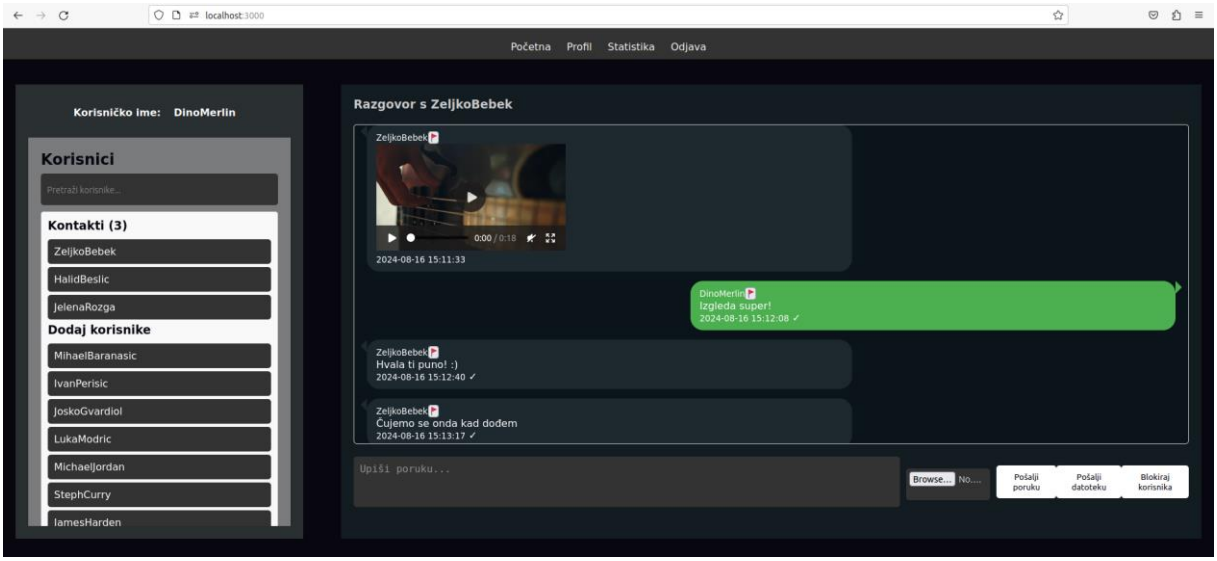
6.4. Stranice aplikacije

Kako bismo imali bolji pregled aplikacije, u ovom poglavlju će biti predstavljene najbitniji dijelovi ChatApp aplikacije.

6.4.1. Prijava i registracija

Prilikom pokretanja aplikacije na početku je prikazana forma za prijavu i poveznica koja vodi na formu za registraciju korisnika. Svaki unos u formu se prvo na klijentskoj strani aplikacije provjerava sa regularnim izrazima kako bi se ograničavala sloboda unosa korisnika. Podaci koji se upisuju putem tih dviju formi se preko HTTP protokola šalju serveru (koji je trenutno localhost), te server provjerava unesene podatke. Ako podaci koji su poslani na server prolaze validaciju, korisnik je registriran ili prijavljen u sustav ovisno o formi koju je posalo.

6.4.2. Glavna stranica



Slika 2: Glavna stranica

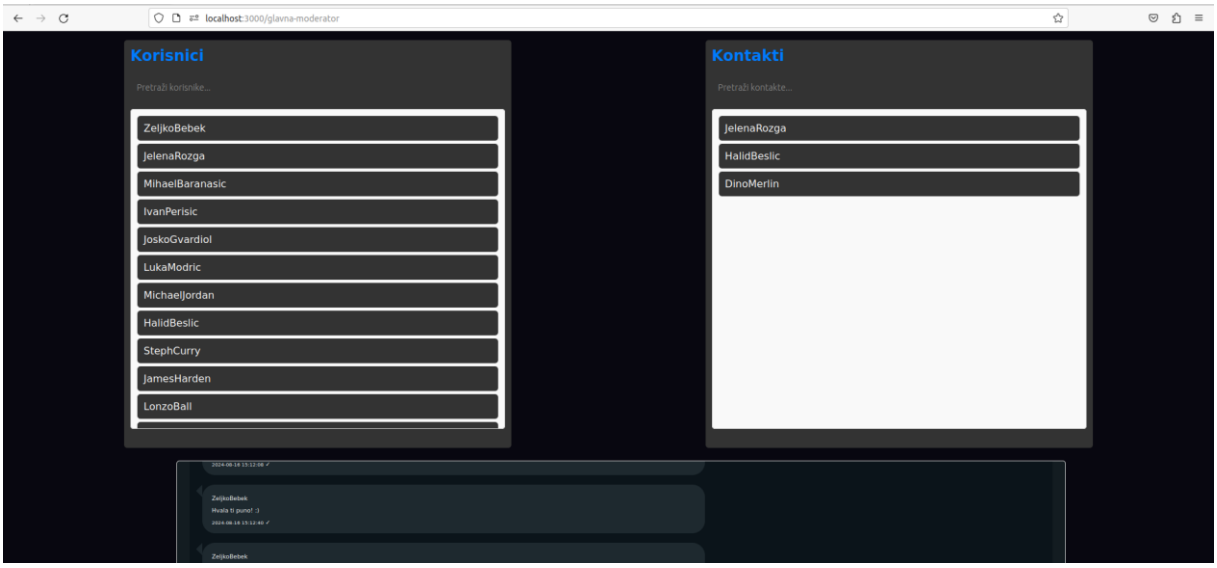
Na slici 4 prikazana je glavna stranica aplikacije koja korisniku nakon što je prijavljen omogućava dodavanje kontakata i otvaranje razgovora sa korisnikovim kontaktima. Kada

korisnik otvori razgovor sa nekim od svojih kontakata, on može tom korisniku poslati poruku ili datoteku koja će biti prikazana u odgovarajućem formatu (npr. .png kao slika, .mp4 kao video i .mp3 kao zvučni zapis). Poruke i datoteke šalju se serveru kroz HTTP protokol kako bi ih server pohranio u bazu podataka. Također se poruke i datoteke mogu prijaviti moderatoru klikom na gumb zastavice koja otvara formu za prijavu poruke.

6.4.3.Profil korisnika

Na profilu korisnika dostupne su opcije za ažuriranje osobnih podataka i postavki obavijesti. Korisnik može odabrati na koji način želi primati obavijesti, birajući između opcija pop-up prozora, prikaza na dashboardu ili putem e-maila. Također, korisnik ima pristup popisu blokiranih korisnika, gdje može vidjeti sve blokirane korisnike i odblokirati ih po potrebi. Pored toga, postoji popis zaprimljenih datoteka koje korisnik može pretraživati prema imenu pošiljatelja ili nazivu datoteke. Tu je i opcija za brisanje korisničkog računa, pri čemu se brišu sve korisničke poruke i datoteke. S obzirom na osjetljivost ovih podataka, njihovo slanje i pohrana trebaju se osigurati putem HTTPS protokola umjesto trenutno korištenog HTTP-a.

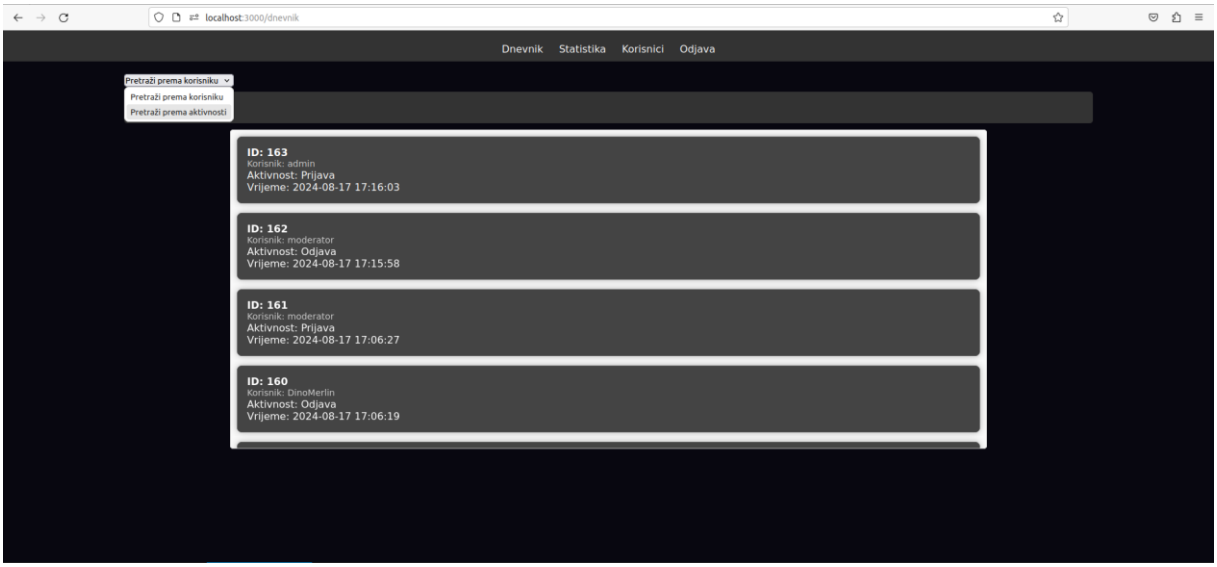
6.4.4.Glavna stranica – Moderator



Slika 3: Glavna stranica – Moderator

Na slici 6 prikazana je glavna stranica za moderatore. Ova stranica dostupna je isključivo korisnicima koji u bazi podataka imaju uloga_id postavljen na 2, što odgovara ulozi moderatora. Moderatorima je omogućeno pretraživanje korisnika i njihovih kontakata, kao i pregled njihovih razgovora. Na temelju pregleda sadržaja razgovora, moderatori mogu ukloniti pojedinačne poruke ili čitav razgovor u slučaju da sadrži neprihvatljiv sadržaj.

6.4.5.Dnevnik aktivnosti



Slika 4: Dnevnik aktivnosti

Na slici 7 je prikazana stranica kojoj ima pristup samo administrator aplikacije. Dnevnik aktivnosti prikazuje sve zabilježene aktivnosti korisnika, uključujući prijavu, odjavu, registraciju, te slanje poruka i datoteka. Osim pregleda dnevnika, administrator može pregledavati sve korisnike i po potrebi ih izbrisati iz baze podataka u slučaju kršenja pravila aplikacije. Također, administrator ima pristup statističkim podacima koji uključuju broj registriranih korisnika, ukupni broj poslanih poruka i datoteka, kao i grafički prikaz broja poslanih poruka i datoteka u zadnjih 10 dana. Ovi podaci omogućuju administratoru da prati korištenje aplikacije i reagira na nepravilnosti ili preopterećenja sustava.

6.4.6. Svi korisnici

Ova stranica je dostupna isključivo administratoru aplikacije. Administrator ima pristup sveobuhvatnom pregledu svih korisnika koji koriste aplikaciju, uključujući mogućnost pretraživanja korisnika prema korisničkim imenima. U slučaju da neki korisnik učestalo krši pravila aplikacije, administrator može jednostavno izvršiti brisanje korisnika, zajedno s njegovim porukama i datotekama, putem jednostavnog gumba, nakon čega je potrebna potvrda za dovršavanje operacije.

6.5. Server.js

Datoteka server.js predstavlja središnji dio ove Node.js aplikacije. Ona djeluje kao glavna datoteka koja postavlja okruženje za rad aplikacije, pokreće servere i definira sve

ključne putanje za dohvat, slanje i brisanje podataka u bazi podataka. Ova datoteka integrira više modula koji omogućuju rad aplikacije, poput express, HTTP, i WebSocket servera. Sve ove komponente zajedno osiguravaju funkcionalnost aplikacije, omogućujući komunikaciju između korisnika u stvarnom vremenu putem WebSocket protokola.

Express server u ovoj aplikaciji služi kao osnova za rukovanje HTTP zahtjevima i usmjeravanje prometa unutar aplikacije. Putem express modula postavljeni su svi API endpointovi koji omogućuju interakciju s bazom podataka, poput kreiranja novih korisnika, slanja i dohvaćanja poruka, kao i upravljanja datotekama. Također, implementirana je i sesija pomoću modula express-session, koja omogućava praćenje korisničkih prijava i autorizacije tijekom korištenja aplikacije [29].

HTTP server služi za rukovanje osnovnim HTTP zahtjevima, omogućavajući pristup aplikaciji putem web preglednika. Ovaj server je posebno važan za omogućavanje pristupa korisnicima, jer služi kao most između korisnika i aplikacije, usmjeravajući njihove zahtjeve prema odgovarajućim resursima i modulima unutar aplikacije [29].

WebSocket server je ključna komponenta za omogućavanje komunikacije u stvarnom vremenu. Korištenjem WebSocket protokola, server omogućava dvosmjernu komunikaciju između klijenta i servera, što je ključno za aplikacije koje zahtijevaju trenutnu razmjenu poruka ili podataka bez potrebe za stalnim osvježavanjem stranice. U ovoj aplikaciji, WebSocket server je zadužen za upravljanje dolaznim porukama i datotekama, obavještavanje korisnika o novim porukama te slanje obavijesti putem e-maila ako je to potrebno [30].

U datoteci server.js definirani su i moduli poput FetchUpravitelj, restKorisnik, restKontakt, restPoruka, restDatoteka, restDnevnik i restStatistika, koji služe za razne operacije vezane uz bazu podataka. Na primjer, restKorisnik modul upravlja operacijama vezanim uz korisnike, poput dohvaćanja, kreiranja, ažuriranja i brisanja korisničkih podataka, dok restPoruka modul upravlja operacijama vezanim uz poruke.

Glavni zadatak ove datoteke je uspostavljanje funkcionalne aplikacije koja može rukovati više korisničkih zahtjeva istovremeno, održavajući stabilnost i sigurnost svih transakcija. Zbog toga je važno osigurati da je ova datoteka postavljena na HTTPS protokol, čime se osigurava sigurnost prijenosa podataka između korisnika i servera.

Kod za postavljanje Express, HTTP i WebSocket servera prikazan je u nastavku:

```
const express = require('express');
const sesija = require('express-session');
const WebSocket = require('ws');
const http = require('http');
```

```
// Kreiranje express servera
const server = express();
const port = 3000;
// Kreiranje HTTP servera
const httpServer = http.createServer(server);
// Kreiranje WebSocket servera
const wss = new WebSocket.Server({ server: httpServer });
// Ostatak datoteke sadrži logiku za rukovanje zahtjevima, definiranje ruta i upravljanje bazom podataka
```

Kao što je vidljivo, server.js datoteka je kritična za rad cijele aplikacije jer postavlja sve osnovne funkcionalnosti koje omogućuju rad aplikacije i komunikaciju između različitih modula i baza podataka. Posebno, u ovoj datoteci integriran je WebSocket server koji omogućuje dvosmjernu komunikaciju između klijenta i servera, što je ključno za aplikacije koje zahtijevaju trenutnu razmjenu podataka, poput aplikacije ChatApp. Ova aplikacija koristi WebSocket za prijenos poruka u stvarnom vremenu, osiguravajući da korisnici mogu odmah vidjeti nove poruke u razgovorima bez potrebe za osvježavanjem stranice. Ovaj aspekt aplikacije je dodatno poboljšan funkcionalnostima koje omogućuju prikazivanje poruka i datoteka u korisničkom sučelju, uključujući vizualni prikaz čije su poruke poslane od strane korisnika.

6.5.1. Prikaz i obrada poruka

Prikaz i obrada poruka na glavnoj stranici ChatApp aplikacije uključuje nekoliko ključnih funkcionalnosti koje omogućuju pravilno dohvaćanje i prikaz poruka i datoteka između korisnika i njegovih kontakata. Dvije glavne funkcije koje omogućuju točan redoslijed poruka i pravilno prikazivanje datoteka su ucitajPoruke i prikaziPorukeIDatoteke.

```
async function ucitajPoruke() {
    // Dohvaća poruke i datoteke između trenutnog korisnika i kontakta iz baze podataka
    let korime = document.getElementById('korime').innerHTML;
    let parametri = {
        method: "GET",
        headers: {
            "Content-Type": "application/json",
        }
    };
    let odgovor = await fetch(`${url}/baza/poruke/${korime}/${trenutniKontakt}`, parametri);
    let poruke = await odgovor.json();
```

```
            // Sličan postupak za dohvat datoteka
            let datoteke = await
fetch(`${url}/baza/datoteke/${korime}/${trenutniKontakt}`, parametri);
            let datotekeData = await datoteke.json();

            // Prikazuje poruke i datoteke zajedno
            prikaziPorukeIDatoteke(poruke, datotekeData);
            scrollajNaKraj();
        }
```

Funkcija ucitajPoruke koristi se za dohvaćanje svih poruka i datoteka koje su razmijenjene između trenutnog korisnika i odabranog kontakta. Ova funkcija koristi JavaScript fetch API za slanje HTTP GET zahtjeva prema serveru, čime se dohvaćaju podaci iz baze.

- Parametri zahtjeva: U sklopu funkcije definiraju se parametri za zahtjev, uključujući metodu (GET) i zaglavlje koje specificira da se radi s JSON podacima.
- Dohvaćanje podataka: Koristeći fetch, funkcija dohvaća poruke i datoteke između korisnika i kontakta. Podaci se zatim pretvaraju u JSON format kako bi se mogli obraditi unutar aplikacije.
- Prikaz poruka i datoteka: Nakon što se podaci dohvatili, funkcija prikaziPorukeIDatoteke se poziva kako bi prikazala poruke i datoteke u korisničkom sučelju. Funkcija scrollajNaKraj osigurava da se pregled poruka automatski pomakne na zadnju poruku.

Funkcija prikaziPorukeIDatoteke:

```
        function prikaziPorukeIDatoteke(poruke, datoteke) {
            // Sortira i prikazuje poruke i datoteke u korisničkom sučelju
            let svePorukeIDatoteke = [...poruke, ...datoteke];
            svePorukeIDatoteke.sort((a, b) => new Date(a.vrijemeSlanja ||
a.vrijemePrimitka) - new Date(b.vrijemeSlanja || b.vrijemePrimitka));
            let html = "";
            for (let item of svePorukeIDatoteke) {
                let liClass = item.korime ===
document.getElementById('korime').innerHTML ? 'moja-poruka' : '';
                if (item.sadrzaj) {
                    html += `<li
class="${liClass}"><small>${item.korime}</small><br>${item.sadrzaj}<br><sma
ll>${item.vrijemeSlanja}</small></li>`;
                } else if (item.naziv) {
                    // Ovdje se prikazuju različiti tipovi datoteka (slike,
video, audio)
```

```
                let fileHTML = '<a href="' + item.putanja + '">' +
item.naziv + '</a>';
                html += `<li
class="${liClass}"><small>${item.korime}</small><br>${fileHTML}<br><small>$
{item.vrijemePrimitka}</small></li>`;
            }
        }
        document.getElementById('listaPoruka').innerHTML = html;
    }
```

Funkcija prikaziPorukeIDatoteke je zadužena za prikazivanje poruka i datoteka u korisničkom sučelju, te osigurava da se sve informacije prikazuju kronološki i u odgovarajućem formatu.

- Kombiniranje poruka i datoteka: Sve poruke i datoteke se kombiniraju u jedan niz (svePorukeIDatoteke) koji se zatim sortira prema vremenu slanja ili primitka. Ova sortirana lista osigurava da se sve stavke prikazuju točno prema redoslijedu kojim su poslane ili primljene.
- Generiranje HTML-a: Funkcija generira HTML kod koji će se ubaciti u element listaPoruka. Prilikom generiranja, provjerava se je li poruku ili datoteku poslao trenutni korisnik ili njegov kontakt, te se dodjeljuje odgovarajuća klasa (moja-poruka ili prazna klasa). Za poruke se prikazuje njihov sadržaj, dok se za datoteke generira link prema datoteci.
- Prikazivanje u sučelju: Generirani HTML kod se zatim dodaje unutar elementa listaPoruka, čime se poruke i datoteke prikazuju korisniku.

6.5.2. Spremanje i brisanje datoteka

Osim što se informacije o datotekama pohranjuju u bazi podataka, također se one spremaju unutar mape uploads kako bi bile dostupne za kasniji dohvat i manipulaciju. U nastavku su objašnjeni relevantni dijelovi koda koji omogućuju ovu funkcionalnost.

Kada korisnik želi poslati datoteku, pokreće se funkcija posaljiDatoteku() u JavaScriptu na klijentskoj strani:

```
async function posaljiDatoteku() {
    let korime = document.getElementById('korime').innerHTML;
    let fileInput = document.getElementById('datoteka');
    let file = fileInput.files[0];
    if(file == undefined) return;
    let formData = new FormData();
    formData.append('datoteka', file);
```

```
            formData.append('posiljatelj', korime);
            formData.append('primatelj', trenutniKontakt);
            let xhr = new XMLHttpRequest();
            xhr.open('POST', `${url}/baza/datoteke`, true);
            xhr.onload = async function () {
                if (xhr.status === 201) {
                    console.log('Datoteka poslana');
                    ws.send(JSON.stringify({ type: 'new_file', posiljatelj:
korime, primatelj: trenutniKontakt, naziv: file.name, putanja:
xhr.responseURL }));
                    fileInput.value = '';
                    await fetch(`https://localhost:3100/baza/dnevnik`, {
                        method: 'POST',
                        headers: { 'Content-Type': 'application/json' },
                        body: JSON.stringify({
                            aktivnost: 'Poslana datoteka korisniku '+
trenutniKontakt,
                            korisnik: korime
                        })
                    });
                } else {
                    console.error('Greška kod slanja datoteke');
                }
            };
            xhr.onerror = function () {
                console.error('Greška kod slanja datoteke');
            };
            xhr.send(formData);
        }
```

Ova funkcija prikuplja datoteku koju je korisnik odabrao, te kreira FormData objekt koji sadrži datoteku i dodatne podatke (poput korisničkog imena pošiljatelja i primatelja). Zatim se pomoću XMLHttpRequest objekta ova forma šalje na server.

Na serveru, datoteke se obrađuju pomoću multer modula koji omogućuje jednostavno rukovanje multipart/form-data zahtjevima. Konfiguracija multera definira mjesto spremanja datoteka na disk:

```
        const storage = multer.diskStorage({
            destination: (req, file, cb) => {
                cb(null, './uploads/');
            },
```

```
        filename: (req, file, cb) => {
            const originalName = file.originalname;
            const fileExt = path.extname(originalName);
            const baseName = path.basename(originalName, fileExt);
            const timestamp = new Date().toISOString().replace(/T/, '
').replace(/\..+/, '');
            const posiljatelj = req.session.korisnik.korime;
            const newFilename = `${baseName}-${posiljatelj}-
${timestamp.replace(/:/g, '-')}${fileExt}`;
            cb(null, newFilename);
        }
    });
    const upload = multer({ storage: storage });
```

Ovdje je definirano da će se datoteke spremati u mapu uploads, a imena datoteka će biti prilagođena tako da sadrže ime pošiljatelja i vremensku oznaku, kako bi se izbjegli eventualni sukobi imena.

Nakon što je datoteka uspješno spremljena na disk, njene informacije se pohranjuju u bazu podataka. Za to je zadužena metoda posaljiDatoteku iz klase DatotekaDAO koja koristi SQL kod za upisivanje podataka u bazu. Ova metoda preuzima ime datoteke (naziv), njen put na disku (putanja), te imena pošiljatelja i primatelja, te sprema ove informacije u bazu podataka. Prvo dohvaća ID-eve korisnika na temelju njihovih korisničkih imena, a zatim te informacije koristi za unos novog zapisa u tablicu datoteka.

Naravno kada se datoteka želi obrisati ona se prvo briše iz baze podataka, ali se prije toga sprema njezina putanja u varijablu, te se ta putanja koristi kako bi se datoteka izbrisala iz diska:

```
    function obrisiDatotekuSaDiska(putanjaDatoteke) {
        return new Promise((resolve, reject) => {
            fs.unlink(path.join(__dirname, '../../', putanjaDatoteke),
(error) => {
                if (error) {
                    console.error('Greška pri brisanju datoteke:',
error);
                    reject(new Error('Greška pri brisanju datoteke'));
                } else {
                    resolve();
                }
            });
        });}
```

6.6. Demonstracija presretanja poruka bez HTTPS-a

Kako bismo demonstrirali nesigurnost aplikacije i pokazali koliko je lako presresti poruke koje se šalju preko HTTP protokola, aplikacija je postavljena na virtualnom stroju sa sustavom Linux Mint, koji se temelji na Ubuntu operacijskom sustavu. Aplikacija će biti pokrenuta na localhost serveru, što znači da zahtjevi neće biti slani na širu mrežu, već će ostati unutar računala na kojem je server pokrenut. Ovaj localhost promet će se zatim snimiti pomoću alata tcpdump, a snimka će biti pohranjena u .pcap formatu kako bi se promet kasnije mogao detaljno analizirati u aplikaciji Wireshark.

Ovo presretanje zahtjeva moguće je napraviti na svakom računalu kroz koje prolazi zahtjev od klijenta do servera (poslužitelja) kao što je to opisano u poglavlju TLS/SSL.

6.6.1. Hvatanje prometa (tcpdump)

Tcpdump je moćan alat za analizu mrežnog prometa koji omogućava korisnicima snimanje i pregledavanje paketa koji prolaze kroz mrežno sučelje. U ovom slučaju, koristit ćemo tcpdump za snimanje prometa na localhostu, koji koristi loopback sučelje (lo). Korištenje alata je napravljeno prema web stranici TCPDUMP & LIBCAP [31]:

Na Linux sustavu, tcpdump instaliramo korištenjem sljedeće naredbe u terminalu:

```
sudo apt-get install tcpdump
```

Nakon instalacije, tcpdump možete koristiti za snimanje prometa na loopback sučelju (lo). Možemo pohraniti promet za kasniju analizu u Wiresharku u .pcap datoteku:

```
sudo tcpdump -i lo -w loopback.pcap
```

-i označava sučelje koje želimo snimati, a -w (eng. write) označava datoteku u koju snimamo promet. Nakon snimanja, datoteku loopback.pcap možemo otvoriti u Wiresharku kako bismo detaljno analizirali mrežni promet.

Nakon pokretanja tcpdump-a potrebno je pokrenuti Node.js poslužitelj i pokrenuti aplikaciju. Zatim trebamo napraviti nekoliko radnji unutar aplikacije kao što je registracija, prijava, slanje poruke i datoteke prije isključivanja tcpdumpa i spremanja paketa u datoteku .pcap.

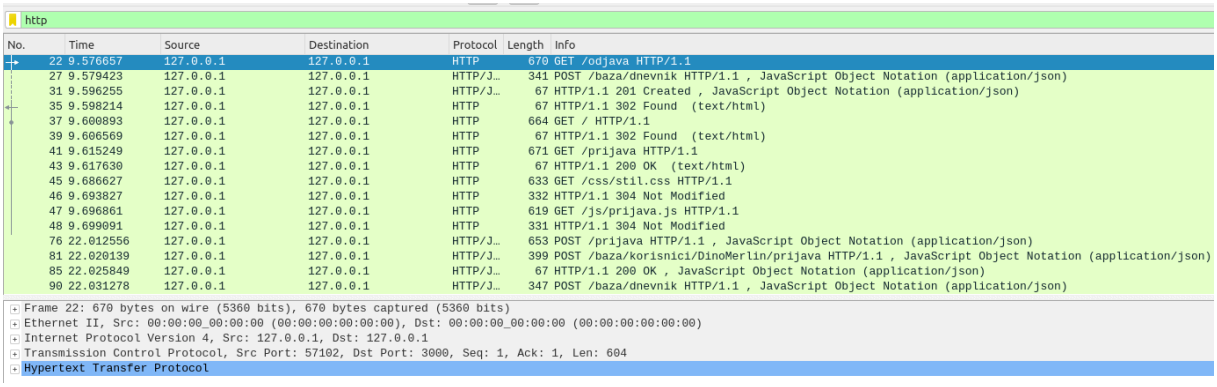
6.6.2.Wireshark

Wireshark je vodeći svjetski alat za analizu mrežnog prometa, široko korišten od strane mrežnih administratora, sigurnosnih stručnjaka, ali i istraživača i entuzijasta. Prema njegovoj glavnoj dokumentaciji Wireshark [32] njegova glavna svrha je snimanje i interaktivna analiza podataka koji se prenose putem mreže u stvarnom vremenu. Wireshark omogućava dubinsko ispitivanje stotina različitih protokola te pruža mogućnost filtriranja, sortiranja i detaljnog pregleda mrežnih paketa, čime olakšava identifikaciju problema ili sigurnosnih propusta u mreži.

Kao open-source alat, Wireshark je besplatan za preuzimanje i korištenje, a njegova popularnost proizlazi iz fleksibilnosti i snage koju nudi u usporedbi s drugim alatima za mrežnu analizu. Pomoću Wiresharka korisnici mogu vizualizirati mrežni promet, pretraživati specifične pakete i analizirati komunikaciju između uređaja na mreži. Zbog svoje opsežne funkcionalnosti i podrške za brojne mrežne protokole, Wireshark je postao standardni alat u arsenalu mnogih profesionalaca koji se bave mrežnim sigurnošću i administracijom [32].

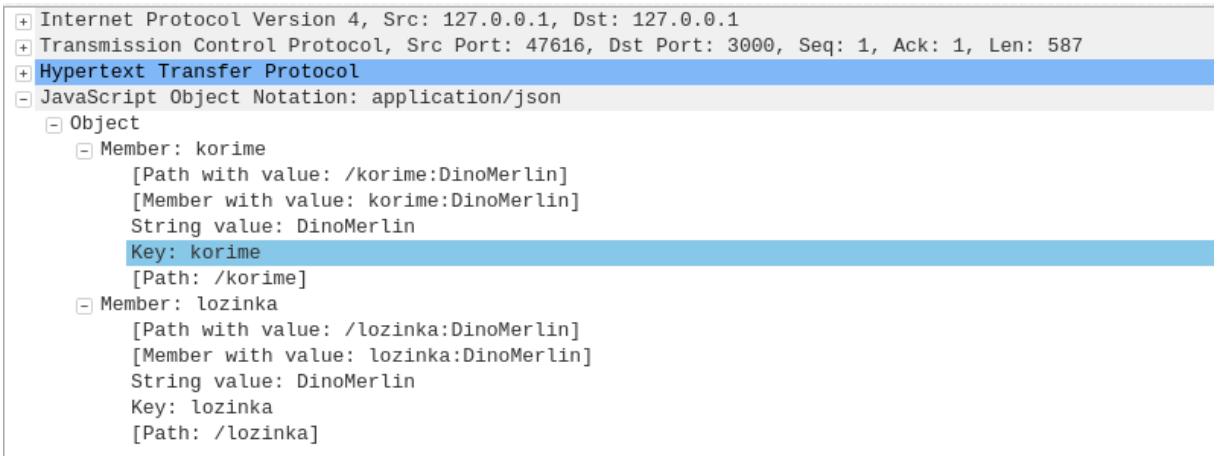
6.6.3.Pregled prometa

Kada smo snimili promet pomoću tcpdump alata možemo otvoriti datoteku pomoću Wireshark-a. U daljnjem primjeru presretanje HTTP zahtjeva će biti bazirano na prijavi korisnika.



Slika 5: Pregled prometa sa Wiresharkom

Kao što je vidljivo na slici 9, tcpdump je ulovio nekoliko paketa sa localhost mreže. Kako bismo pronašli što nas zanima možemo u filter staviti HTTP protokol. Kada filtriramo pakete za HTTP protokol moramo pronaći paket koji sadrži POST metodu i /prijava u info. U ovom slučaju to je paket broj 76.



Slika 6: Podaci prijave

Kao što je vidljivo na slici 10, unutar paketa prikazanog u Wiresharku nalazi se segment „JavaScript Object Notation“ (JSON) koji sadrži podatke o korisničkoj prijavi u čitljivom tekstualnom formatu. Ovdje je jasno vidljivo koje je korisničko ime i lozinku korisnik unio tijekom prijave. Ova informacija lako može biti iskorištena od strane zlonamjernog napadača, koji se uz ove podatke može prijaviti kao korisnik DinoMerlin (u ovom primjeru). Time bi napadač stekao pristup svim razgovorima korisnika, s mogućnošću slanja vlastitih poruka ili datoteka kontaktima korisnika. Takve datoteke mogu sadržavati zlonamjerni softver, što predstavlja ozbiljan sigurnosni rizik, jer se sve to može dogoditi bez znanja i dopuštenja legitimnog korisnika.

6.7. Prijelaz na HTTPS

U ovom poglavlju bit će opisan postupak generiranja kriptografskih ključeva i self-signed SSL certifikata, koji će se koristiti za osiguranje Node.js servera putem HTTPS-a.

6.7.1. Generiranje ključeva i certifikata

Self-signed certifikati, iako nisu pouzdani za produkcijske okoline, idealni su za razvojne i testne svrhe jer omogućuju postavljanje sigurnosnog sloja bez dodatnih troškova.

Koraci generiranja ključeva i certifikata prema PheonixNAP i SSL.com [33], [34]:

1. Instalacija OpenSSL-a

Prvi korak u generiranju ključeva i certifikata je instalacija OpenSSL-a, alata koji se koristi za upravljanje kriptografskim funkcijama. Na većini Linux distribucija, uključujući Ubuntu i Linux Mint, OpenSSL je već predinstaliran. U suprotnom, može se instalirati sljedećom naredbom:

```
sudo apt-get install openssl
```

2. Generiranje privatnog ključa

Privatni ključ predstavlja temelj sigurnosti servera jer se koristi za dešifriranje podataka koji su šifrirani javnim ključem. Za generiranje privatnog ključa koristimo sljedeću naredbu:

```
openssl genrsa -out privatekey.pem 2048
```

3. Kreiranje zahtjeva za certifikat (CSR)

Zahtjev za potpisivanje certifikata (Certificate Signing Request - CSR) je datoteka koja sadrži informacije o organizaciji i javnom ključu. Ovaj zahtjev se obično šalje certificirajućem tijelu (CA) koje izdaje potpisani certifikat. U slučaju self-signed certifikata, ovaj korak je koristan za postavljanje osnovnih informacija. Za generiranje CSR-a koristimo sljedeću naredbu:

```
openssl req -new -key privatekey.pem -out server.csr
```

Prilikom pokretanja ove naredbe OpenSSL nas traži da unesemo podatke kao što su ime organizacije, država i zajedničko ime (Common Name - CN), koje obično predstavlja domenu ili IP adresu servera.

4. Generiranje self-signed certifikata

Nakon kreiranja CSR-a, možemo generirati self-signed certifikat. Ovaj certifikat se koristi za šifriranje podataka između klijenta i servera, ali kako je opisano u poglavlju „Certifikati“ neće biti prepoznat kao pouzdan od strane preglednika jer nije potpisan od ovlaštenog certificirajućeg tijela. Za generiranje certifikata koristimo sljedeću naredbu:

```
openssl x509 -req -days 365 -in server.csr -signkey privatekey.pem -out server.crt
```

Ova naredba kreira certifikat server.crt koji vrijedi 365 dana i koji je potpisan privatnim ključem privatekey.pem.

Za potrebe razvoja i testiranja, možemo pojednostaviti generiranje certifikata pomoću jedne naredbe koja će obuhvatiti sve prethodne korake:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout privatekey.pem -out server.crt
```

Ova naredba generira privatni ključ i self-signed certifikat u jednoj naredbi, što ubrzava proces postavljanja sigurnosnog sloja za testne svrhe.

Prema OpenSSL man stranici, ovo su značenja dijelova naredbi [35]:

- openssl: Ovo je naziv alata koji se koristi za upravljanje kriptografskim operacijama. OpenSSL je široko korišten alat za generiranje ključeva, certifikata, enkripciju, i dekripciju podataka.
- req: Ova opcija specificira da koristimo "certificate request and certificate generating utility". To je alat unutar OpenSSL-a koji omogućuje generiranje zahtjeva za certifikat (CSR) ili, u ovom slučaju, self-signed certifikata.
- -x509: Ova opcija označava da želimo generirati X.509 certifikat, koji je standardni format za certifikate. U kombinaciji s req, ova opcija omogućuje generiranje self-signed certifikata bez potrebe za vanjskim certificirajućim tijelom (CA).
- -nodes: Ova opcija stoji za "no DES" i ukazuje da privatni ključ ne treba biti šifriran lozinkom. Ako se ova opcija izostavi, OpenSSL će tražiti lozinku svaki put kada se koristi privatni ključ, što može biti nezgodno za automatizirane ili razvojne svrhe.
- -days 365: Ova opcija postavlja valjanost certifikata na 365 dana. Možete prilagoditi broj dana prema potrebi. Nakon isteka tog roka, certifikat će postati nevažeći i bit će potrebno generirati novi.

- -newkey rsa:2048: Ova opcija kombinira generiranje novog RSA ključa s duljinom ključa od 2048 bita. RSA (Rivest–Shamir–Adleman) je jedan od najčešće korištenih algoritama za kriptografske ključeve. Veličina od 2048 bita nudi dobar balans između sigurnosti i performansi.
- -keyout privatekey.pem: Ova opcija specificira naziv datoteke u koju će biti pohranjen generirani privatni ključ. U ovom primjeru, ključ će biti spremljen u datoteku pod nazivom privatekey.pem.
- -out server.crt: Ova opcija definira naziv datoteke u koju će biti pohranjen generirani self-signed certifikat. U ovom slučaju, certifikat će biti pohranjen u datoteku server.crt.

Kada pokrenemo naredbu dobijemo nekoliko pitanja koja možemo, ali i ne moramo sve popuniti. Evo primjera popunjavanja certifikata:

```
      You are about to be asked to enter information that will be incorporated
      into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
      There are quite a few fields but you can leave some blank
      For some fields there will be a default value,
      If you enter '.', the field will be left blank.
      -----
      Country Name (2 letter code) [AU]:HR
      State or Province Name (full name) [Some-State]:Varaždinska županija
      Locality Name (eg, city) []:Varaždin
      Organization Name (eg, company) [Internet Widgits Pty Ltd]:Fakultet
organizacije i informatike
      Organizational Unit Name (eg, section) []:
      Common Name (e.g. server FQDN or YOUR name) []:localhost:3000
      Email Address []:primjer@gmail.com
```

6.7.2. Implementacija HTTPS-a na Node.js server

Nakon što smo generirali privatni ključ i certifikat, potrebno ih je implementirati na Node.js server. Prvo je potrebno dodati datoteke ključa i certifikata u mapu u kojoj se nalazi aplikacija, odnosno server. U Node.js aplikaciji potrebno je koristiti https modul umjesto http modula.

Na početku je potrebno dodati https i fs (eng. File sistem modul za čitanje datoteka) modul na server:

```
const https = require('https');
const fs = require('fs');
```

Zatim generiran ključ i certifikat dodajemo u postavke HTTPS servera:

```
const httpsPostavke = {
  key: fs.readFileSync('./privatekey.pem'),
  cert: fs.readFileSync('./server.crt')
};
```

Zatim moramo postaviti httpsServer sa tim postavkama, te također premijestiti WebSocekt server na httpsServer kako bi i komunikacija u stvarnom vremenu bila sigurna:

```
const httpsServer = https.createServer(httpsPostavke, server);
const wss = new WebSocket.Server({ server: httpsServer });
```

Na kraju pokrećemo server pomoću metode httpsServer.listen i zadajemo port serveru:

```
httpsServer.listen(3100, () => {
  console.log('HTTPS server pokrenut na portu: https://localhost:3100');
});
```

S obzirom da smo postavili server na HTTPS, više nije dostupan putem HTTP-a. Kako bismo osigurali da svi korisnici budu preusmjereni na HTTPS, možemo postaviti HTTP server koji će automatski preusmjeravati sve HTTP zahtjeve na HTTPS. To će osigurati da korisnici koji pokušaju pristupiti aplikaciji putem HTTP-a budu preusmjereni na sigurnu HTTPS vezu.

```
const httpRedirectServer = express();

httpRedirectServer.use((zahtjev, odgovor) => {
  res.redirect(`https://${zahtjev.hostname}${odgovor.url}`);
});

httpRedirectServer.listen(port, () => {
  console.log('HTTP server za preusmjeravanje pokrenut na portu 3000');
});
```

Cijeli kod bi morao izgledati slično ovome:

```
const express = require('express');
const sesija = require('express-session');
const WebSocket = require('ws');
const https = require('https');
const fs = require('fs');
```

```
//ostatak dodanih modula
const server = express();
const port = 3000;
const httpsPostavke = {
  key: fs.readFileSync('./privatekey.pem'),
  cert: fs.readFileSync('./server.crt')
};
//putanje i ostali dijelovi servera
const httpsServer = https.createServer(httpsPostavke, server);
const wss = new WebSocket.Server({ server: httpsServer });
// logika postavljanja WebSocket servera
const httpRedirectServer = express();
httpRedirectServer.use((zahtjev, odgovor) => {
  const targetUrl = `https://${zahtjev.hostname}:3100`;
  console.log(`Preusmjeravanje: ${targetUrl}`);
  odgovor.redirect(targetUrl);
});
httpRedirectServer.listen(3000, () => {
  console.log('HTTP server za preusmjeravanje pokrenut na portu 3000');
});
httpsServer.listen(3100, () => {
  console.log('HTTPS server pokrenut na portu: https://localhost:3100');
});
```

Osim što moramo promijeniti putanje na serveru, moramo paziti da na našem klijentskom JavaScriptu i ostalim backend datotekama pristupamo preko točne putanje, tj. preko HTTPS protokola. Nekoliko primjera što moramo ispraviti:

Zbog toga što sada koristimo sigurnu verziju WebSocket-a, moramo na klijentskom kodu postaviti taj protokol ispravno.

```
window.addEventListener('load', async () => {
    postaviWebSocket();
});
function postaviWebSocket() {
    ws = new WebSocket('ws://localhost:3000');
    // ostatak funkcije
}
```

Ruta definirana u funkciji postaviWebSocket treba biti promijenjena u:

```
window.addEventListener('load', async () => {
    postaviWebSocket();
```

```
});
function postaviWebSocket() {
    ws = new WebSocket('wss://localhost:3100');
    // ostatak funkcije
}
```

Osim WebSocket-a mijenjamo i fetch zahtjeve u klijentskom kodu:

Iz: `let url = "http://localhost:3000";`

```
async function posaljiFormu(korime, lozinka) {
       // priprema za fetch
        let odgovor = await fetch(url + '/prijava', {
            // postavke parametara zahtjeva
        });
      // ostatak koda
}
```

U: `let url = "http://localhost:3000";`

```
// ostatak koda
```

6.7.3. Dodatni sigurnosni sloj HSTS

Kako je rečeno u HSTS poglavlju, HTTP Strict Transport Security (HSTS) predstavlja dodatni sigurnosni sloj koji osigurava da web preglednici komuniciraju isključivo putem sigurnih HTTPS veza. HSTS pomaže u sprječavanju napada presretanja, poznatih kao "man-in-the-middle" napadi, osiguravajući da komunikacija ostane šifrirana.

Za implementaciju HSTS-a u Node.js aplikaciji koja koristi Express framework, često se koristi helmet modul. Prema službenoj npm dokumentaciji [36], helmet je kolekcija middleware funkcija koje pomažu u zaštiti Express aplikacija postavljanjem različitih HTTP zaglavlja. Jedan od tih middleware-a je helmet.hsts, koji omogućuje jednostavnu primjenu HSTS-a u aplikaciji.

Kod za implementaciju HSTS-a:

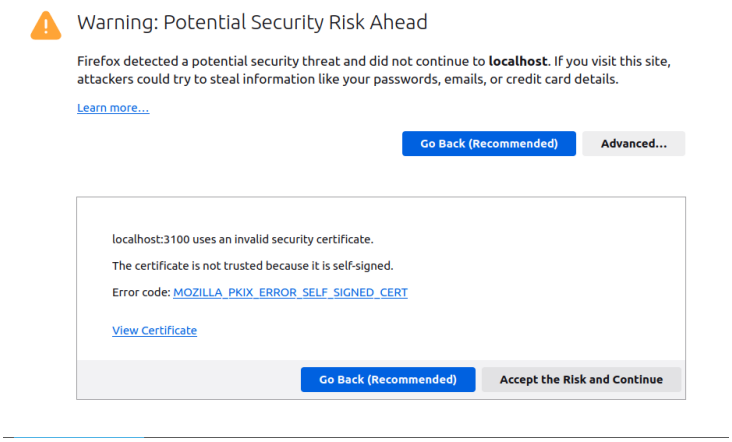
```
const helmet = require('helmet');


server.use(helmet.hsts({
  maxAge: 31536000, // Vrijeme u sekundama za koje će preglednik zapamtiti da koristi samo HTTPS
  includeSubDomains: true, // Primjenjuje HSTS na sve poddomene
  preload: true // Omogućava uključivanje stranice u HSTS preload listu
}));
```

U ovom kodu, maxAge određuje koliko dugo (u sekundama) preglednik treba pamtiti da koristi HTTPS za komunikaciju s stranicom. Vrijednost od 31536000 sekundi odgovara jednoj godini, što je preporučeno trajanje za HSTS. Opcija includeSubDomains osigurava da HSTS politika vrijedi i za sve poddomene, čime se dodatno povećava sigurnost. Postavka preload omogućava da domena bude uključena na listu HSTS preload, koja se koristi u modernim preglednicima kako bi se osiguralo da određene web stranice uvijek koriste HTTPS [36].

Implementacija HSTS-a putem helmet modula omogućuje da komunikacija između servera i klijenta bude sigurna, sprječavajući potencijalne sigurnosne prijetnje. Integracija HSTS-a na ovaj način dodaje značajan sloj sigurnosti, čineći aplikacije otpornijima na napade.

6.7.4.Provjera rada HTTPS-a i HSTS-a

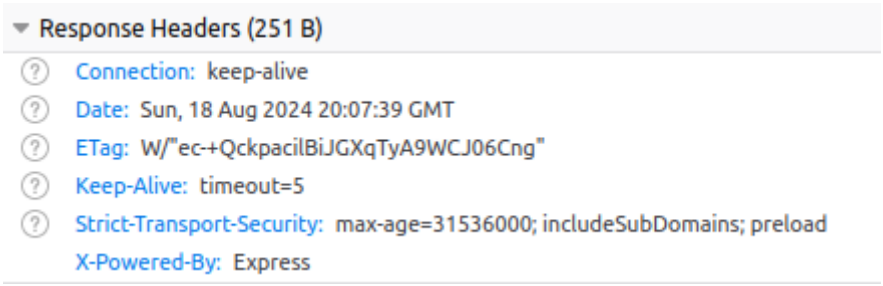
Kako bismo provjerili radi li aplikacija sada kada je preusmjerena sa HTTP-a na HTTPS, možemo započeti sa upisivanjem web adrese http://localhost:3000 i provjeriti da li se adresa prebacuje na https://localhost:3100. Prvo pokrećemo server sa naredbom: `NODE_TLS_REJECT_UNAUTHORIZED='0' node server.js`. Ova naredba nam omogućava privremeno rješenje pokretanja servera sa certifikatom kojeg nije potpisala CA, jer Node.js po defaultu ne dopušta neautorizirane certifikate.



Slika 7: Provjera HTTPS-a

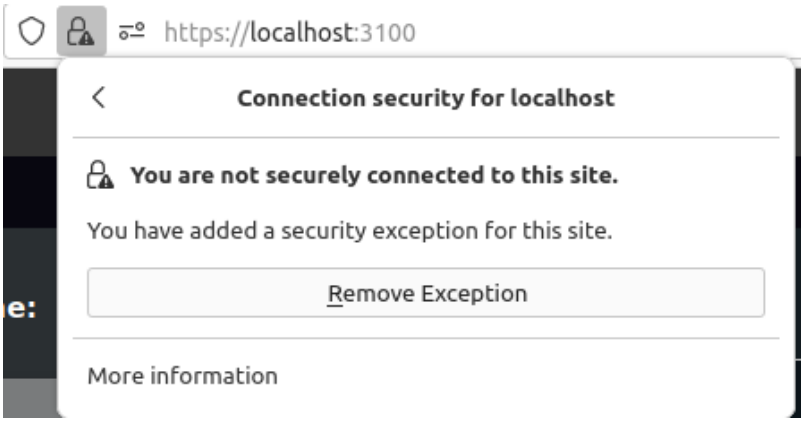
Kao što je vidljivo na slici 11, server je prebacio web adresu na https://localhost:3100. Također dobijemo upozorenje o potencijalnom riziku sigurnosti. To je zbog toga što aplikacija koristi self-signed certifikat koji web pretraživač ne smatra sigurnim. Ovo upozorenje možemo zaobići klikom na „Prihvaćam rizike i nastavi“. Ovakav pristup je uredu koristiti kod testiranja i razvoja, ali kada se aplikacija postavlja na mrežu morao bi se koristiti valjan certifikat.

Točnu konfiguraciju HSTS modula možemo provjeriti pomoću „Inspect element“ na kartici „Network“. Ako trenutno na kartici nije ništa prikazano potrebno je osvježiti stranicu. Zatim klikom na jedan od HTTPS zahtjeva možemo vidjeti atribut Strict-Transport-Security sadrži podatke koje smo mi postavili, što je vidljivo na slici 12.



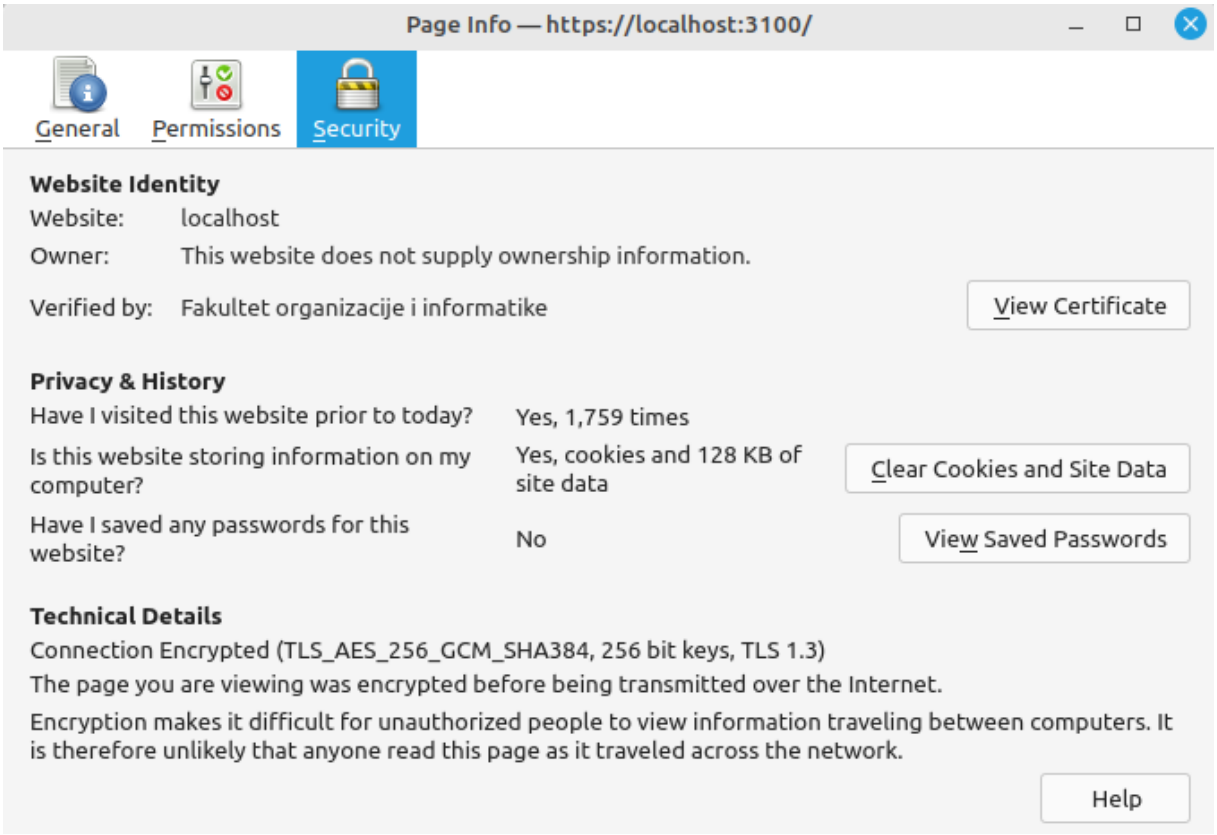
Slika 8: Provjera HSTS-a

Na kraju još možemo i provjeriti da se na stranici nalazi i naš generirani certifikat. To možemo klikom na slijedeći gumb prikazan na slici 13:



Slika 9: Provjera certifikata

Klikom na „More information“ na slici 13 otvara nam se dijaloški okvir koji nam tko je verificirao certifikat i gumb za pregled certifikata vidljiv na slici 14.

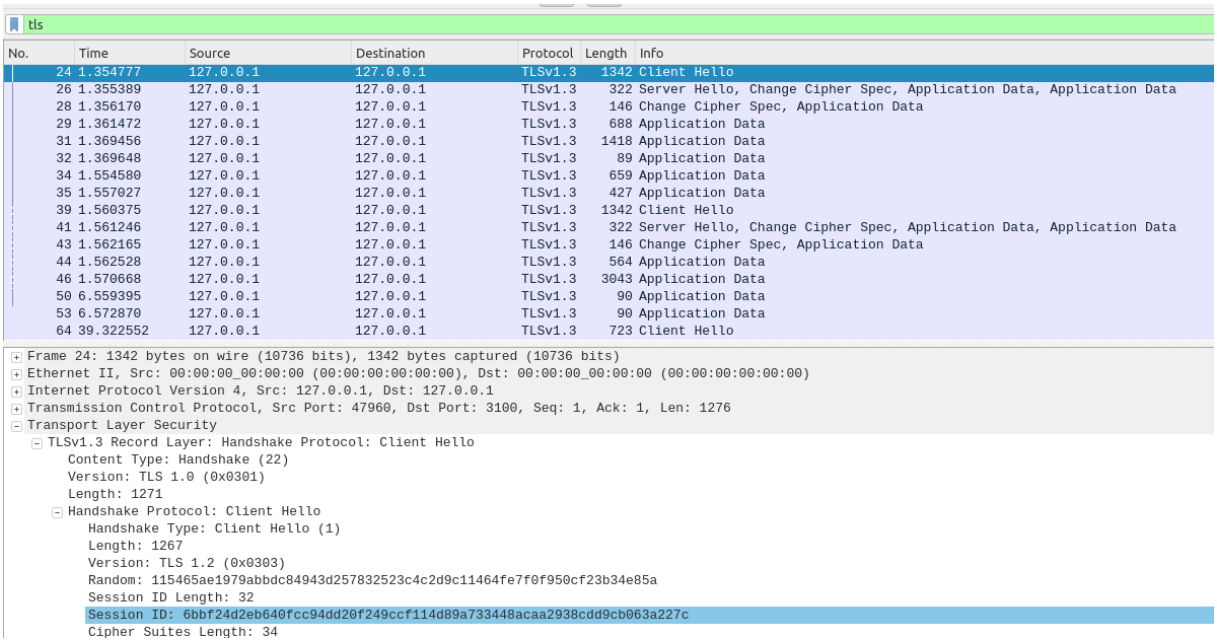


Slika 10: Pregled certifikata

6.8. Sigurnost podataka sa HTTPS-om

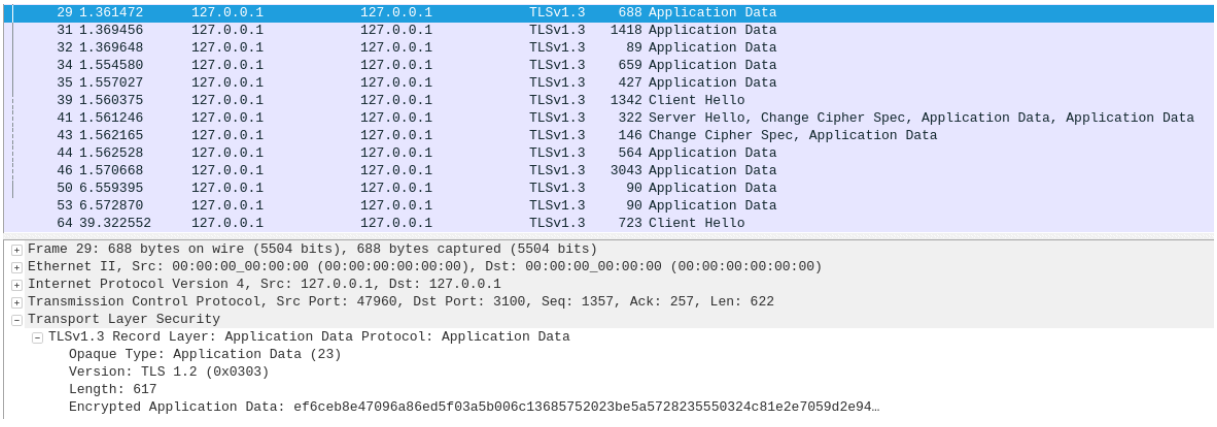
Na isti način kao i kod poglavlja „Demonstracija presretanja poruka bez HTTPS-a“ možemo snimiti promet uz pomoć tcpdump i pregledati pakete pomoću alata Wireshark.

Pokrenemo naredbu `sudo tcpdump -i lo -w loopback.pcap` i snimimo nekoliko aktivnosti na aplikaciji kao što su prijava, registracija, slanje poruke. Zatim tu datoteku otvaramo u Wiresharku.



Slika 11: Wireshark sa HTTPS

Kao što je vidljivo sa slike 15, Wireshark ne filtrira pakete direktno po HTTPS protokolu, već po TLSv1.3 protokolu, pa je u filter potrebno unijeti TLS. Vidljivo je kako TLS koristi protokol rukovanja za postavljanje veze između klijenta i servera. Također kako je kod HTTP paketa vidljivo u „Info“ stupcu koji zahtjevi koriste koju metodu (GET, POST, PUT…), kod TLS-a to nije vidljivo pa ne možemo znati koji paket sadrži kriptirane podatke o prijavi. Jedino što možemo shvatiti iz priložene slike 15 je da „Application data“ sadrži podatke koji se šalju u aplikaciji. Otvaranjem tog paketa dobijemo slijedeći prikaz na slici 16:



Slika 12: Prikaz kriptiranih podataka

Na slici 16 prikazan je način na koji TLS protokol omogućava slanje kriptiranih podataka aplikacije, zajedno s verzijom protokola i duljinom zahtjeva. Ovim se postupkom podaci učinkovito skrivaju od potencijalnih napadača, čime se značajno otežava njihov pokušaj dešifriranja i pristupa osjetljivim informacijama.

7. Zaključak

Na temelju detaljne analize važnosti i razvoja HTTPS-a, te sigurnosnih protokola kao što su TLS i HSTS, može se zaključiti da su ovi elementi ključni za modernu web sigurnost. Kroz rad se istražila evolucija HTTPS-a, prikazane su razlike između HTTP-a i HTTPS-a, te su analizirane potencijalne prijetnje koje prijete bez korištenja enkripcije. Usporedbom različitih kriptografskih algoritama, istaknute su njihove snage i slabosti, čime je dodatno objašnjen izbor odgovarajućeg sigurnosnog pristupa u raznim kontekstima.

Korištenje SSL certifikata ne samo da osigurava enkripciju podataka, već potvrđuje identitet servera, čime se povećava povjerenje korisnika. Implementacija ovih certifikata, kao i nadogradnja na HTTPS, pruža zaštitu od napada poput "Man in the Middle". Dodatni sigurnosni sloj koji pruža HSTS osigurava da se komunikacija odvija isključivo putem HTTPS-a, što dodatno otežava napadačima presretanje i manipulaciju podacima.

U praktičnom dijelu rada, demonstrirana je implementacija HTTPS-a na Node.js serveru, uključujući generiranje ključeva i certifikata, te prilagodbu servera za rad sa sigurnosnim protokolima. Ovaj dio jasno pokazuje kako se teorijske spoznaje mogu primijeniti u stvarnim okruženjima.

Zaključno, HTTPS se nameće kao neophodan standard za osiguranje sigurnosti podataka na webu, a uvođenje naprednih protokola kao što su HSTS i ažuriranje TLS konfiguracija dodatno učvršćuju sigurnost aplikacija. Uz kontinuirano praćenje sigurnosnih prijetnji i primjenu najboljih praksi, HTTPS osigurava visoke standarde zaštite podataka, što je presudno za očuvanje povjerenja korisnika i integriteta sustava.

Literatura

[1] E. Quinton, *Safety of web applications : risks, encryption and handling vulnerabilities with PHP*, 1st Edition. London: ISTE Press - Elsevier, 2017.

[2] „Networking 101: Transport Layer Security (TLS) - High Performance Browser Networking (O'Reilly)". Pristupljeno: 21. srpanj 2024. [Na internetu]. Dostupno na: https://hpbn.co/transport-layer-security-tls/

[3] „RFC 9112 - HTTP/1.1". Pristupljeno: 25. srpanj 2024. [Na internetu]. Dostupno na: https://datatracker.ietf.org/doc/html/rfc9112

[4] „RFC 2818 - HTTP Over TLS". Pristupljeno: 07. kolovoz 2024. [Na internetu]. Dostupno na: https://datatracker.ietf.org/doc/html/rfc2818

[5] B. Pollard, *HTTP/2 in action*. New York: Manning Publications, 2019.

[6] „The HTTP Protocol As Implemented In W3". Pristupljeno: 30. srpanj 2024. [Na internetu]. Dostupno na: https://www.w3.org/Protocols/HTTP/AsImplemented.html

[7] „RFC 1945 - Hypertext Transfer Protocol -- HTTP/1.0". Pristupljeno: 30. srpanj 2024. [Na internetu]. Dostupno na: https://datatracker.ietf.org/doc/html/rfc1945#section-5

[8] „RFC 9113 - HTTP/2". Pristupljeno: 08. kolovoz 2024. [Na internetu]. Dostupno na: https://datatracker.ietf.org/doc/html/rfc9113#name-http-2-protocol-overview

[9] „HTTP vs. HTTPS: What's the Difference? - Rublon". Pristupljeno: 16. srpanj 2024. [Na internetu]. Dostupno na: https://rublon.com/blog/http-vs-https-whats-the-difference/

[10] „RFC 6797 - HTTP Strict Transport Security (HSTS)". Pristupljeno: 09. kolovoz 2024. [Na internetu]. Dostupno na: https://datatracker.ietf.org/doc/html/rfc6797

[11] N. D. Jorstad, „Cryptographic algorithm metrics", sij. 1997.

[12] O. Goldreich, *Foundations of Cryptography*. Cambridge University Press, 2001.

[13] „What is cryptography or a Cryptographic Algorithm? | DigiCert FAQ". Pristupljeno: 16. srpanj 2024. [Na internetu]. Dostupno na: https://www.digicert.com/faq/cryptography/what-is-cryptography-or-a-cryptographic-algorithm

[14] R. Sheldon, P. Loshin, i M. Cobb, „What is Encryption and How Does it Work? | Definition from TechTarget". Pristupljeno: 18. srpanj 2024. [Na internetu]. Dostupno na: https://www.techtarget.com/searchsecurity/definition/encryption

[15] M. Alam Hossain, M. Biddut Hossain, M. Shafin Uddin, i S. Md Imtiaz, „Performance Analysis of Different Cryptography Algorithms“, 2016. [Na internetu]. Dostupno na: www.ijarcsse.com

[16] L. Van Houtven, „Crypto 101 lvh“, 2013.

[17] M. Vanitha i R. Mangayarkarasi, „Comparative Study of Different Cryptographic Algorithms“, *Journal of Information Security*, sv. 11, izd. 3, str. 138–148, lip. 2020, doi: 10.4236/JIS.2020.113009.

[18] Rolf. Oppliger, *SSL and TLS : theory and practice*. Artech House, 2016.

[19] S. Thomas, *SSL and TLS Essentials Securing the Web*. Wiley, 2000.

[20] D. Wagner i B. Schneier, „Analysis of the SSL 3.0 Protocol“, 1996.

[21] R. Oppliger, „SSL and TLS: Theory and practice“, str. 257, 2009.

[22] „RFC 6101 - The Secure Sockets Layer (SSL) Protocol Version 3.0“. Pristupljeno: 08. kolovoz 2024. [Na internetu]. Dostupno na: https://datatracker.ietf.org/doc/html/rfc6101#page-5

[23] H. Krawczyk, K. G. Paterson, i H. Wee, „On the Security of the TLS Protocol: A Systematic Analysis“, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, sv. 8042 LNCS, izd. PART 1, str. 429–448, 2013, doi: 10.1007/978-3-642-40041-4_24.

[24] „RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3“. Pristupljeno: 08. kolovoz 2024. [Na internetu]. Dostupno na: https://datatracker.ietf.org/doc/html/rfc8446

[25] „RFC 9608 - No Revocation Available for X.509 Public Key Certificates“. Pristupljeno: 09. kolovoz 2024. [Na internetu]. Dostupno na: https://datatracker.ietf.org/doc/html/rfc9608

[26] M. Zalewski, *The tangled Web: a guide to securing modern Web applications*. No Starch Press, 2011.

[27] Bruce. Potter i Bob. Fleck, *802.11 security*. O’Reilly, 2003.

[28] G. Kassaras, *Hacker’s Elusive Thoughts The Web*. Man In The Middle Ltd, 2016.

[29] „npm | Home“. Pristupljeno: 17. kolovoz 2024. [Na internetu]. Dostupno na: https://www.npmjs.com/

[30] „WebSocket - Web APIs | MDN“. Pristupljeno: 17. kolovoz 2024. [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

[31] „Home | TCPDUMP & LIBPCAP“. Pristupljeno: 17. kolovoz 2024. [Na internetu]. Dostupno na: https://www.tcpdump.org/

[32] „Wireshark · Go Deep“. Pristupljeno: 17. kolovoz 2024. [Na internetu]. Dostupno na: https://www.wireshark.org/

[33] „Manually Generate a Certificate Signing Request (CSR) Using OpenSSL - SSL.com“. Pristupljeno: 18. kolovoz 2024. [Na internetu]. Dostupno na: https://www.ssl.com/how-to/manually-generate-a-certificate-signing-request-csr-using-openssl/

[34] „How to Generate CSR With OpenSSL | phoenixNAP KB“. Pristupljeno: 18. kolovoz 2024. [Na internetu]. Dostupno na: https://phoenixnap.com/kb/generate-openssl-certificate-signing-request

[35] „openssl(1): OpenSSL tool - Linux man page“. Pristupljeno: 18. kolovoz 2024. [Na internetu]. Dostupno na: https://linux.die.net/man/1/openssl

[36] „helmet - npm“. Pristupljeno: 18. kolovoz 2024. [Na internetu]. Dostupno na: https://www.npmjs.com/package/helmet

Popis slika

Slika 1: ERA model baze podataka 25
Slika 2: Glavna stranica 27
Slika 3: Glavna stranica – Moderator 28
Slika 4: Dnevnik aktivnosti 29
Slika 5: Pregled prometa sa Wiresharkom 37
Slika 6: Podaci prijave 38
Slika 7: Provjera HTTPS-a 45
Slika 8: Provjera HSTS-a 46
Slika 9: Provjera certifikata 46
Slika 10: Pregled certifikata 47
Slika 11: Wireshark sa HTTPS 48
Slika 12: Prikaz kriptiranih podataka 48

Popis tablica

Tablica 1: Kriptoalgoritmi [15] 14

Prilozi

Prilog 1: kod aplikacije

- Kompletan kod implementirane aplikacije dostupan je na sustavu FOI radovi u zip arhivi. Potrebno je pri pokretanju aplikacije izraditi .env datoteku unutar mape "moduli" i u njoj postaviti svoje Gmail korisničko ime i lozinku. Aplikacija se pokreće pomoću naredbe npm start, a prije toga potrebno pokrenuti npm run pripremi.